Firelight Technologies FMOD Ex
Introduction

The FMOD Ex sound system is an audio engine for game developers, multimedia developers, sound designers, musicians and audio engineers, based on the years of experience in audio technology. It also aims high - to push the boundaries of audio implementation for games and the like while at the same time using minimal resources and being scalable.

Some of the most exciting features, which are described in more detail later are:

- **Suite of built in DSP special effects** which do not rely on any platform or operating system. (for a 100% cross platform audio experience). Includes **high quality I3DL2 compatible reverb**!
- **Next-gen console support**. PS3, Xbox 360 and Wii are fully supported.
- **Sound designer focus and tool**. The new suite of tools and functionality means FMOD is usable by sound designers and musicians and not just programmers. Sound authors will have the ability to create complex audio models and tweak them in real-time over the network (or even internet) while the game/application is still running!
- **Full 3D sound support** including linear/nonlinear/custom rolloff models, multiple listener support, occlusion and obstruction, sound cones, and support for stereo or multichannel samples being played in 3d!.
- **Geometry occlusion engine**. You can supply FMOD with a polygon scene and it will automatically occlude and obstruct direct path and reverb signals for you!
- **Virtual voices** to allow a game to play thousands of sounds at once on limited hardware without worrying about handling the logic to switch sounds off and on themselves.
- Support for **over 20 file formats**.
- **Advanced streaming engine** supporting gapless stitching/sentencing of sounds, low cpu overhead, multiple stream support, over-ridable file callbacks and more.
- **Compressed sample playback**. ADPCM, MPEG and XMA are able to be stored in memory without decompressing or streaming them, as if they were normal static samples!
- **Sub mixing** and channel groups.
- **2D / 3D sound morphing.** Set up a user supplied 5.1 or 7.1 2D mix, and morph between it and directional 3D sound! Great for entering and leaving volumetric sound sources.
- **Advanced 'DSP network' based software engine** to rival the most complex software synthesizer packages, all performed in real-time while the game/application is running! **Matrix panning** allows sound channels to be mapped to any speaker in any combination.
- **Object oriented API** supporting C, C++ and C#.
- **Plug-in support** for ultimate flexibility. FMOD and VST plugins are supported. Everything in FMOD Ex has been designed with future expansion in mind.
- **SIMD optimized** (ie SSE, VMX, VFPU, ALTIVEC) mixing and filter routines for low cpu overhead. It is faster to use FMOD's software mixer than go through the driver overhead of DirectSound!
FMOD Ex supports the following hardware platforms. No other audio system available supports this many platforms.

- Microsoft Windows 32/64bit.
- Linux 32/64bit.
- Macintosh OS X 32/64bit.
- Sony PlayStation 2, Playstation 3, Playstation Portable, PlayStation Vita
- Microsoft Xbox 360.
- Nintendo Wii, 3DS, Wii U.
- Apple iOS devices (iPhone, iPod Touch, iPad).
- Android
- Google Native Client

That's 14 platforms with more coming! No other game audio library can claim to match anywhere near that many platforms!
Feature list

Unified API.

Samples, streams, music and CD access are accessed via a 'Sound' class
All types of sounds, including mods, midi files, wavs, oggs, samples, streams, cd
tracks and fsb files can be accessed seamlessly through the one API.

Virtual Channels

Virtual channels allow thousands of channels to play on limited
hardware/software. Voices are swapped in and out according to 3d distance and priority.

**Plug-in System**

New file formats, output modes, and encoders can be added or downloaded by the user as DLLs.
VST and Winamp DSP plug-in support for effects is included.

**Digital CD Playback**

Digital CDDA playback allows dsp effects / spectrum analysis, ripping etc just as if it was a normal PCM file being played back.

**C++ API**

In FMOD Ex, a C++ API is available as well as a standard C API. All FMOD API features are accessible through simple class types, such as the system class, sound class, channel class, DSP class. C/C++ headers naming conventions closely mapped.
For example - FMOD::System::init() in the C++ header would become FMOD_System_Init() in the C header.

**C# and Visual Basic API**

FMOD Ex has full support for managed C# and Visual Basic interfaces

**Multiple simultaneous soundcard support**

FMOD 3 was limited by only supporting 1 sound card at a time, so if you wanted to output to multiple cards at once you would have to instance fmod.dll multiple times.
Multiple output at once support is simply done by initializing multiple 'System' objects.

**Multi-speaker output support**

Now FMOD has a full multichannel mixer, even 2D sounds can be played in 5.1 (or 7.1!). Sounds can even swap their channel assignments around so left and
right of a stereo sound are swapped around, mixed or all placed in the rear left speaker for example. The way this is available is FMOD supports pan matrices. Any input sound channel can be redirected to any output speaker, and on top of this percentages/fractional levels are supported, so there are no absolute speaker assignments.

Via ASIO, FMOD Ex now also supports full multichannel output access to up to 16 output channels for high end sound devices.

**Multi-speaker input support**

Multichannel wav's, oggs and FSB files are supported for 5.1 music for example.

**Low latency recording support**

FMOD Ex supports super low latency recording, processing and output. Via ASIO the recording->DSP->playback latency can be as low as 1-3ms! This is great for realtime processing and playback of recorded audio.

**Enhanced Internet features**

- Internet audio streaming. Custom internet streaming code is included, which allows for seamless SHOUTcast, Icecast and http streaming support.

- Download capability. A side effect of the FMOD™ modular file system which supports network files, even static samples can be loaded off the internet. In fact you can use the FMOD™ API to write an arbitrary file downloader!

**File format support**

FMOD currently supports a wide range of audio file formats.
- AIFF - (Audio Interchange File Format)  
- ASF - (Advanced Streaming format, includes support for the audio tracks in video streams)  
- ASX - (playlist format - contains links to other audio files. To access contents, the FMOD Ex tag API is used)  
- DLS - (DownLoadable Sound format for midi playback. Can also be used as a
stand alone container format in FMOD)
- FLAC - (Lossless compression codec)
- FSB - (FMOD sample bank format generated by FSBank and FMOD designer tool)
- IT - (Impulse tracker sequenced mod format. FMOD Ex also fully supports resonant filters in .IT files, and the per channel or per instrument echo effect send, that can be enabled in ModPlug Tracker. This is cross platform effect support and does not require DirectX like other libraries do.)
- M3U - (playlist format - contains links to other audio files. To access contents, the FMOD Ex tag API is used)
- MID - MIDI using operating system or custom DLS patches.
- MOD - (Protracker / Fasttracker and others sequenced mod format)
- MP2 - (MPEG I/II Layer 2)
- MP3 - (MPEG I/II Layer 3, including VBR support)
- OGG - (Ogg Vorbis format)
- PLS - (playlist format - contains links to other audio files. To access contents, the FMOD Ex tag API is used)
- RAW - (Raw file format support. The user can specify the number of channels, bitdepth, format etc)
- S3M - (ScreamTracker 3 sequenced mod format)
- VAG - (PS2 / PSP format, playable on all platforms!)
- WAV - (Microsoft Wave files, including compressed wav files. PCM, MP3 and IMA ADPCM compressed wav files are supported across all platforms in FMOD Ex, and other compression formats are supported via windows codecs on that platform).
- WAX - (playlist format - contains links to other audio files. To access contents, the FMOD Ex tag API is used)
- WMA - (Windows Media Audio format)
- XM - (FastTracker 2 sequenced format)
- XMA - (Xbox 360 only)
- XWMA - (Xbox 360 only)
- AudioQueue - (Apple iOS devices only, supports AAC, ALAC and MP3 decoding in software and hardware)

File format plugins are also supported so the number of formats supported is limitless!
Note AAC is not included in FMOD Ex because the only reference source for this is GPL and FMOD Ex does not contain GPL protected code. To support this a user may add their own plugin to support it externally.
Wav Writer output

All output can be written to a wav file, and with encoder plug-ins, it can even be encoded in real-time to MP3 or other file formats!

Sample accurate seeking

Most systems seek to a compression block boundary such as mp3 which decodes in blocks of 1152 samples at a time. FMOD Ex supports sample accurate seeking and decoding. For example you could seek to sample offset 1,000,000 exactly, and extract 1 sample of audio. This accuracy is good for DJ type programs that need to sync streams properly.

Enhanced streaming engine

A low latency stream decoder spreads the decode burden over time instead of doing it in chunks (cpu spikes!) is included. This means smoother frame-rates in game.

Enhanced sample format support

24bit, 32bit integer and 32bit IEEE float sample support is included. Alongside standard mono/stereo sample support, now multi-channel sample support is included! Wav, ogg and user created sounds are examples of sound formats that support multi-channel sound.

Advanced mixing engine

- Enhanced output channel support
  Most systems only allow mixing to mono or stereo output. FMOD Ex allows mixing to any number of output channels, for example 6 channel output (with panning) to allow for 5.1 or Dolby digital output in real-time for 3d sound! Stereo and 5.1 are optimized as a special case fast-path for extra speed.
- Full DSP data flow network based mixing engine.
  New mixing routines with separate resample/mix/effects stages. This is a node based multiple input/output DSP engine which is extremely
flexible and allows submixing, splitting and advanced speaker location and selection.

- **High quality mixing**
  All mixing is floating point with full 32bit interpolation.
  Resampling modes supported are
  - No interpolation
  - Linear interpolation
  - Cubic interpolation
  - 5 point spline interpolation!
  All resampling is done with true 32bit precision using a 32bit fractional, it is not downsampled or compromised in any way.

- **Matrix Panning**
  Sounds can have their input channels mapped to any output channel through a simple 2D matrix. For example the left and right parts of a stereo sound can be positioned anywhere in a 5.1 speaker array, in any combination, in one speaker, or all speakers. It is totally flexible.

- **Volume ramping**
  Linear volume ramps between pan/volume changes are included as standard. This removes clicks in sound that changes pan or volume frequently.

### 3D Sound enhancements

- **Rolloff models.**
  Logarithmic, linear, or custom rolloff models supported (per voice).
- **Geometry API.**
  A revolutionary step up in audio realism is supported with FMOD Ex's custom geometry engine. This allows polygon scenes to be added to FMOD so that it can automatically calculate obstruction/occlusion as the user moves around the world.
- **Multiple listener support.**
  Multiple 3d listeners for split screen support are supported.
- **Sound cone support.**
  Sound cones are supported to give sounds direction.
- **Stereo / multichannel sound support.**
  Stereo samples or even multichannel samples can be positioned in 3D, with their component channels (ie left/right parts of a stereo sound) positioned in 3D space, configurable by the user.
- **3D / 2D morphing.**
  Now sounds can morph between being totally 3d directional point sources, and
discrete 2D sources with speaker levels set by the user! This is great for entering and leaving a volumetric sound source. As an example, a stereo 3D sound can morph between being a directional point source, to a stereo 2D sound that envelopes you, then back again.

**User delay on sound playback**

A new 'setDelay' function is available so a sound can be specified to start after a certain period of time (samples or ms) - can be called between init and start on a channel

**MIDI Support**

FMOD Ex includes its own software midi playback, so that midi playback works cross platform. Patch sets / DLS banks have to currently be provided with the song, or FMOD Ex will take advantage of any found in the operating system.

**Stitching / sentencing**

Seamless stitching, for sounds allows one sound to end then another starts immediately afterwards without gaps. This is great for commentary or interactive music.

**Built in software based special effects.**

FMOD Ex hosts a whole suite of special effects surpassing any system available considering it will work on every platform FMOD supports. Here are some of the effects that are supported as default. More can be added through plugins.

- Oscillators - sine, square, saw up, saw down, triangle and noise wave oscillators.
- 3 Low-pass with varying speed/quality.
- High-pass.
- 2 Echo filters.
- Flange.
- Distortion.
- Normalizer.
• Compressor.
• Tremolo.
• Delay.
• Parametric EQ.
• Realtime pitch shifter (changes pitch not playback speed)
• Chorus.
• Freeverb simple reverb.
• SFX high quality I3DL2 compatible reverb.

Channel groups, and submixing.

Multiple channel groups can be created and channels assigned to these groups. From there a variety of commands can be issued on a group such as volume, mute, frequency, pause and more. Master volume can be controlled through the use of a channel group, and multiple channel groups can be used for multiple master volume assignments, which is very useful for things like relative volume of GUI sounds vs in game sounds for example, or music vs special effects volume. This allows greater flexibility in controlling audio levels.

Submixing allows effects to be placed on groups of channels, without affecting other channels. This is an advanced feature which is really useful for saving CPU usage or keeping some sounds dry while others are affected by DSP effects for example.

Enhanced callback support

• 'latency adjusted' or 'real-time' flag for callbacks. This means you can get a callback at mix time, or audible time (the 2 are different, by the length of time determined by the mixer's buffer size)
• sample accurate user timer callbacks (ms or sample based) for global or per channel

Memory and filesystem overrides

FMOD Ex of course allows the user to override FMOD's file and memory system through callbacks.

FMOD Designer tool and API
Sound Designer Tool

This easy to use and flexible sound designer tool allows simple or complex multi-layer/effect/envelope based sound events to be modeled and created by the sound designer. The capabilities would include such things as layering, effects, random behaviour, and stitching of sounds.

The aim is for a sound designer to totally design the in game audio from an external tool, and simply supply the programmer with assets and an event list to implement. If the audio behaviour needs to be changed within the game, it should be up to the sound designer not the programmer to do this.

The layering screen allows for complex audio models (such as a car engine with multiple cross fading channels, sounds and effects) to be totally controlled by the author, then all the programmer has to do is call the previously defined set of simple commands, such as SendEvent and UpdateParameter. In the car model cast, the ‘parameter’ in UpdateParameter might just be ‘revs’ or ‘torque’ or some other English type value, rather than a value defined by a programmer.

FMOD Event API This is an API for programmers to interface to the data produced by the FMOD Designer tool. This API consists of very simple commands such as:

- Init
- Close
- Load
- GetEvent / Start
- UpdateParameter

All event behaviour is specified by the FMOD Designer tool, not the programmer, to make it totally data driven.

Network tweaking features

As part of the sound designer tool, the user can tweak the audio parameters in a game over the network while the game is running! A sound designer
now gets even more control over the outcome of the audio mix by being able to alter sound parameters such as volume / frequency / randomization etc while the game is running. This will save hours of time instead of the usual routine of testing, quitting, tweaking, recompiling, running. Even with that old method it can lead to mistakes which take several attempts to perfect. Using the network tweaking tool the sound designer can get it right first time.
TERMINOLOGY / BASIC CONCEPTS.

Introduction

Throughout FMOD documentation certain terms and concepts will be used. This section will explain some of these to alleviate confusion.

It is recommended when you see an API function highlighted as a link, that you check the API reference for more detail.
Samples vs bytes vs milliseconds

Within FMOD functions you will see references to PCM samples, bytes and milliseconds.  
To understand what the difference is a diagram has been provided to show how raw PCM sample data is stored in FMOD buffers.

![Diagram of 16 bit stereo sample data, 44khz.]

In this diagram you will see that a stereo sound has its left/right data interleaved one after the other.

- A left/right pair (a sound with 2 channels) is called a sample.
- Because this is made up of 16bit data, 1 sample = 4 bytes.
- If the sample rate, or playback rate is 44.1khz, or 44100 samples per second, then 1 sample is $\frac{1}{44100}$th of a second, or $\frac{1}{44}$th of a millisecond. Therefore 44100 samples = 1 second or 1000ms worth of data.

To convert between the different terminologies, the following formulas can be used.

- $ms = samples \times 1000 / samplerate$.
- $samples = ms \times samplerate / 1000$.
- $samplerate = samples \times 1000 / ms$.
- $bytes = samples \times bits / channels$.
- $samples = bytes \times bits / channels$.

Some functions like Sound::getLength provide the length in milliseconds, bytes and samples to avoid needing to do these calculations.
• **Sounds. Samples vs compressed samples vs streams.**

When a sound is loaded, it is either decompressed as a static sample into memory as PCM (samples), loaded into memory in its native format and decompressed at runtime (compressed samples), or streamed and decoded in realtime (in chunks) from an external media such as a harddisk or CD (streams).

• "**Samples**" are good for small sounds that need to be played more than once at a time, for example sound effects. These generally use little or no CPU to play back and can be hardware accelerated. See **FMOD_CREATESAMPLE**.

• "**Streams**" are good for large sounds that are too large to fit into memory and need to be streamed from disk into a small ringbuffer that FMOD manages. These take a small amount of CPU and disk bandwidth based on the file format. For example mp3 takes more cpu power to decode in real-time than a PCM decompressed wav file does. A streaming sound can only be played once, not multiple times due to it only having 1 file handle per stream and 1 ringbuffer to decode into. See **FMOD_CREATESTREAM**.

• "**Compressed samples**" are a new advanced option that allows the user to load a certain compressed file format (such as IMA ADPCM, MP2, MP3 and XMA formats currently), and leave them compressed in memory without decompressing them. They are software mixed on the CPU and don't have the 'once only' limitation of streams. They take more cpu than a standard PCM sample, but actually less than a stream due to not doing any disk access and much smaller memory buffers. See **FMOD_CREATECOMPRESSEDSAMPLE**.

You may notice "Sample" and "Stream" terminology is used here but there is no class name with this terminology in them. That is because all FMOD APIs are now consolidated into one "**Sound**" type.

**By default System::createSound will want to decode the whole sound fully into memory** (ie, as a decompressed sample).
To have it stream in realtime and save memory, use the `FMOD_CREATESTREAM` flag when creating a sound, or use the helper function `System::createStream` which is essentially the same as `System::createSound` but just has the `FMOD_CREATESTREAM` flag added in automatically for you.

To make a compressed sample use `System::createSound` with `FMOD_CREATECOMPRESSEDSAMPLE`. 
Hardware vs Software

FMOD Ex has its support for either hardware accelerated sound playback, via DirectSound or console hardware API's, but FMOD also has its own fallback software mixing mechanism.

**Note:** Only Windows with a high spec soundcard, PS2, PSP, and Wii support FMOD_HARDWARE.

With hardware and software based sounds comes certain features and trade-offs when they are used.

Hardware sounds (created with FMOD_HARDWARE usually have lower CPU impact, have lower latency, and can get access to hardware reverb like EAX4 for example.

Hardware sounds are also limited in some ways, for example due to DirectSound limitations on Windows for example, arbitrary loop points are not supported with static samples (it is either loop the whole sound, or don't loop the sample), and non reverb effects cannot be played on them (ie chorus, distortion, lowpass etc).

Software sounds (created with FMOD_SOFTWARE sometimes have higher CPU impact, but can do much more, for example complex looping, realtime analysis, effects and sample accurate synchronization.

**Hardware vs Software.**

**Hardware Pros.**

- Usually lower latency. (Although on consoles or ASIO output in windows, using FMOD_SOFTWARE can have extremely low latency as low as 2-5ms)
• Less CPU time. (Although on Windows software is a lot faster due to bad hardware sound card driver design, and inefficiencies in the DirectSound API).
• On Windows, access to EAX2, EAX3, EAX4, I3DL2 reverb per voice. (FMOD Ex has its own high quality I3DL2 reverb solution in software, but may not be as flexible or have the quality of EAX4 for example.).
• Free hardware obstruction / occlusion (this is usually equivalent to a lowpass filter or reverb attenuation which can also be performed in software at some expense to the CPU), but only on EAX compatible sound cards on Windows.
• FMOD_SOFTWARE is cross platform.
• On PS2, PSP, XBox, GameCube, Wii, hardware voices can play back ADPCM compressed sound data with no cpu hit.
• On a limited number of soundcards, hardware 3d sounds will be realtime encoded into an AC3 Dolby Digital stream via a digital / optical output on the card so an amplifier can play it in 3D surround sound. FMOD software mixing now supports 5.1 and 7.1 mixing at slightly higher CPU expense, and will work via analog outputs such as soundcards with 3 stereo jacks to run to a 5.1 speaker setup.

Hardware Cons.

• No point to point looping on win32. XBox and GameCube allow point to point looping and PS2 only allows loopstart, so therefore cross platform compatibility cannot be assured.
• No access to hardware effects per voice. Most PC sound cards and consoles do not support hardware accelerated effects such as lowpass, distortion, flange, chorus etc.
• No loop count control. A sound can only be looped infinitely or not at all.
• Inconsistent feature support, for example a PS2 does not support EAX reverb, and 3d sound implementations always sound different.
• Sometimes a lot slower than FMOD software mixing on Windows. Virtual voices that make a lot of state changes when swapping in and out can be very expensive in hardware (noticable framerate drops), but for free in software.

Software Pros.

• Consistent sound on every platform, there is no variation in playback.
• Sample accurate synchronization callbacks and events.
• Compressed sample playback support without using streams.
• Cross platform reverb.
- Complex looping and loop counts.
- Reverse sample playback.
- Spectrum analysis.
- Filters per channel or for the global mix, to perform effects such as lowpass, distortion, flange, chorus etc.
- Complex DSP network construction for realtime sound synthesis.
- Access to final mix buffer to allow analyzing, drawing to screen, or saving to file.

**Software Cons.**

- Latency on some sound devices (such as win32 waveout output) can be high.
- Memory usage is higher due to allocation of mix units and mix buffers, or simply the fact of having to store sounds in main ram rather than sound ram. (becoming less relevant these days).
Channels and sounds.

When you have loaded your sounds, you will want to play them. When you play them you will use System::playSound, which will return you a pointer to a
Channel / FMOD_CHANNEL handle.
The index that System::playSound requires is generally recommended to always be FMOD_CHANNEL_FREE. This will mean FMOD will choose a non playing channel for you to play on.
2D vs 3D.

A 3D sound source is a channel that has a position and a velocity. When a 3D channel is playing, its volume, speaker placement and pitch will be affected.
automatically based on the relation to the **listener**.
A **listener** is the player, or the game camera. It has a position, velocity like a sound **source**, but it also has an **orientation**.

The **listener** and the **source** distance from each other determine the **volume**. The **listener** and the **source** relative velocity determines the **pitch** (doppler effect).
The orientation of the **listener** to the **source** determines the **pan** or **speaker placement**.

A 2D sound is simply different in that it is not affected by the 3D sound **listener**, and does not have doppler or attenuation or speaker placement affected by it. A 2D sound can call **Channel::setSpeakerMix**, **Channel::setSpeakerLevels** or **Channel::setPan**, whereas a 3D sound cannot.
A 3D sound can call any function with the word **3D** in the function name, whereas a 2D sound cannot.

For a more detailed description of 3D sound, read the tutorial in the documentation on 3D sound.
Firelight Technologies FMOD Ex
GETTING STARTED.

Introduction

The FMOD Ex API has been designed to be intuitive and flexible. In this tutorial an introduction to using the engine as well as the key issues involved in using it effectively will be explained.
Set up. What to include and what to link.

See "Platform specific issues" in this documentation to see what files to link into your project to make FMOD Ex function for each platform.

In C/C++, include "fmod.h" if you want to use the C interface only. Include "fmod.hpp" if you want to use the C++ interface. Note that the constants, callbacks, defines and enums are stored within fmod.h, so fmod.hpp includes fmod.h. If you are using C++ you will be interchanging between both.

For Delphi, C# and Visual Basic, you will see equivalent headers to use in your application.
Initialization.

The simplest way to initialize fmod is to simply call `System::init`. That's it. FMOD will set up the soundcard and other factors using default parameters.

When looking at the documentation for `System::init`, remember that the `maxchannels` parameter is the number of simultaneous voices you would like to be played in your game at once. This is nothing to do with how many hardware voices the soundcard may have, or how many software mixed voices there may be available.

These voices are virtual voices. This means you can play as many sounds as you want at once and not worry about the issue of hardware or software resources available.

You can safely play EVERY sound in your game simultaneously without fear of `System::playSound` running out of voices or stealing other playing voices, and for this reason, it is acceptable to set maxchannels to a high number. 1, 100, 200, 1000. It is up to you and your type of title.

Note 1000 voices playing at once does not negatively impact performance because the majority of those will not be audible (non audible voices are 'virtualized'). There is only a small cost in sorting and swapping those voices as the FMOD Ex virtual voice manager controls which voices are heard and which aren't.

Let's have a look at an example of initializing FMOD Ex.

```c
FMOD_RESULT result;
FMOD::System *system;

result = FMOD::System_Create(&system); // Create the main system
if (result != FMOD_OK)
{
    printf("FMOD error! (%d) %s\n", result, FMOD_ErrorString(result))
    exit(-1);
}

result = system->init(100, FMOD_INIT_NORMAL, 0); // Initialize
if (result != FMOD_OK)
{
    printf("FMOD error! (%d) %s\n", result, FMOD_ErrorString(result))
    exit(-1);
}
```
Here we have the most basic setup of the FMOD engine. It will use 100 virtual voices. Note that mod/s3m/xm/it/midi formats use 1 voice when playing. Do not extend the voice count here thinking it will give more voices to these file formats when playing, because they won't. These formats have their own internal pool voices that they use.
Configuration options

The output hardware, FMOD's resource usage, and other types of configuration options can be set if you desire behaviour differing from the default. These are generally called before System::init. The main ones are.

- **System::setOutput** - To choose an alternative output method. For example you can choose between DirectSound, WinMM, ASIO, no-sound, wave-writer or a number of other output options in windows. Each platform will have their own output choices. Don't call this unless you need to. You don't need to call it especially if all you are doing is setting the default. That would be pointless.

- **System::setDriver** - To choose an alternative sound card driver for a particular output mode. This is useful if you have multiple sound cards and want to choose one beside the default. Again, don't bother calling this if all you are doing is setting it to the default. You should enumerate devices with System::getNumDrivers and System::getDriverInfo if you want to give the user the choice.

- **System::setHardwareChannels** - Call this if you want to limit the number of audible hardware voices, or request that a minimum number of hardware voices be available before reverting to 100% software mixed voice support. The 'minimum' option is to guarantee a certain number of voices are audible at once.

- **System::setSoftwareChannels** - Call this if you want to set a different number of audible software mixed voices used by FMOD Channels. This will be purely for polyphony reasons or CPU / memory resource usage reasons. Do not adjust this thinking it will give more voices to mod/s3m/xm/it/midi formats. They do not use this channel pool and have their own internally.

- **System::setSoftwareFormat** - Call this to change settings in the FMOD software mixer. This includes sample rate, output format (ie integer vs float), output channel count (ie for multi-output channel asio devices for example), memory usage and mixing quality.

- **System::setDSPBufferSize** - Call this only if there are issues with stuttering on slow machines or bad soundcard drivers. This will affect software mixing latency, and can have adverse effects if misused. Some titles may want to let the user select between 'low latency' and 'compatible' modes, so they can trade off latency to audible stability by adjusting the buffersize.

- **System::setSpeakerMode** - Call this to set the output speaker mode. This only
affects the FMOD software mixing engine. The default is stereo (5.1 on xbox and xbox360 and 7.1 on ps3), and can be changed if desired. Note speaker modes with higher channel counts leads to higher memory usage.

Here is an example of initializing FMOD with some configuration options. **Remember these options are just that. Optional! Do not call these if you don't need to and don't just cut and paste this code without knowing what it does!** For example you can't just go setting the speaker mode to 5.1 if the user doesn't have a 5.1 speaker system!

```cpp
FMOD_RESULT result;
FMOD::System *system;

result = FMOD::System_Create(&system);  // Create the main system object.
ERRCHECK(result);

result = system->setSpeakerMode(FMOD_SPEAKERMODE_5POINT1);  // Set the output to 5.1.
ERRCHECK(result);

result = system->setSoftwareChannels(100);  // Allow 100 software mixed voices to be audible at once.
ERRCHECK(result);

result = system->setHardwareChannels(32, 64, 32, 64);  // Require the soundcard to have at least 32 2D and 3D hardware voices, and clamp it to using 64 if it has more than this.
ERRCHECK(result);

result = system->init(200, FMOD_INIT_NORMAL, 0);  // Initialize FMOD with 200 virtual voices.
ERRCHECK(result);
```
• **Loading and playing.**

To play the sounds you must load them first! To do this, use `System::createSound` or `System::createStream`. A sound by default will try to decompress the whole sound into memory (if `System::createSound` is used), that is why if the sound is large, it is better to stream it (by using `System::createStream`) which means it will decode at runtime, with a small fixed size memory buffer, and not use the memory a sample would.

For more on this see the [Terminology/Basic Concepts tutorial](#).

Here is an example of loading an mp3 file. By default `System::createSound` will decompress the whole MP3 into 16bit PCM. This could mean the amount of memory used is many times more than the size of the file.

```c
FMOD::Sound *sound;
result = system->createSound("../media/wave.mp3", FMOD_DEFAULT, 0, &sound);
ERRCHECK(result);
```

Here is an example of opening an mp3 file to be streamed. `System::createStream` will open the file, and pre-buffer a small amount of data so that it will be able to play instantly when `System::playSound` is called.

```c
FMOD::Sound *sound;
result = system->createStream("../media/wave.mp3", FMOD_DEFAULT, 0, &sound);
ERRCHECK(result);
```

To specifically make a sound software mixed, you must use `FMOD_SOFTWARE`. This is necessary if you want to use things such as DSP effects, spectrum analysis, getwavedata, point to point looping and other more advanced techniques.

```c
FMOD::Sound *sound;
result = system->createSound("../media/wave.mp3", FMOD_SOFTWARE, 0, &sound);
ERRCHECK(result);
```
Here is an example of loading an mp3 file into memory as a sample, but not decompressing it when it loads, using the use **FMOD_CREATECOMPRESSEDSAMPLE** flag. This will automatically make the sound software mixed if **FMOD_HARDWARE** or **FMOD_SOFTWARE** is not specified. Hardware sound playback cannot support this flag unless the format is ADPCM on Xbox, VAG on PS2/PSP and GCADPCM on Gamecube/Wii. Platforms like PS3 and Xbox 360 are all done on the cpu (usually a different core to the main cpu so it does not affect performance).

```cpp
FMOD::Sound *sound;
result = system->createSound("..\media\wave.mp3", FMOD_CREATECOMPRESSEDSAMPLE);
ERRCHECK(result);
```

**Warning!** This mode is to be used with care. It acts just like a PCM sample, but incurs a heavier CPU cost at runtime. FMOD decodes the sound from its compressed format as it plays it.

Now to play the sound or stream. This is as simple as calling **System::playSound**.

```cpp
FMOD::Channel *channel;
result = system->playSound(FMOD_CHANNEL_FREE, sound, false, &channel);
ERRCHECK(result);
```

This sound is now playing in the background! Your app will continue on from this point.

Things to note about playSound.

- **You do not need to store the channel handle** if you do not want to. That parameter can be 0 or NULL. This is useful if you don't care about updating that instance of the sound, and if it is a one shot sound (ie it does not loop). For example

  ```cpp
  FMOD::Channel *channel;
  result = system->playSound(FMOD_CHANNEL_FREE, sound, false, 0);
  ERRCHECK(result);
  ```

- **You can start the sound paused**, so you can update its attributes without the change being audible. That is what the 'paused' parameter is used for. For example, if you set it to true, set the volume to 0.5, then unpaused it, the sound
would play at half volume. If you had set the paused flag to false and executed
the same logic, you may hear the sound play at full volume for a fraction of a
second. This can be undesirable.

```
FMOD::Channel *channel;
result = system->playSound(FMOD_CHANNEL_FREE, sound, true);
ERRCHECK(result);
result = channel->setVolume(0.5f); // Set the volume while it is paused.
ERRCHECK(result);
result = channel->setPaused(false); // This is where the sound really starts.
ERRCHECK(result);
```

- **A 'channel' is an instance of a sound.** You can play a sound many times at
  once, and each time you play a sound you will get a new channel handle.
  Note that this is only if it is not a stream. Streams can only be played once
  at a time, and if you attempt to play it multiple times, it will simply restart
  the existing stream and return the same handle that it was using before. This
  is because streams only have 1 stream buffer, and 1 file handle. To play a
  stream twice at once, open and play it twice.
- **Always use FMOD_CHANNEL_FREE.** This lets FMOD pick the
  channels for you, meaning that it uses FMOD's channel manager to pick a
  non playing channel. FMOD_CHANNEL_REUSE can be used if the
  desired effect is to pass in an existing channel handle and use that for the
  playsound. It can be used to stop a sound spawning a new instance every
  time System::playSound is called, and only play once at a time.
- **You do not have to 'free' or 'release' a channel handle.** Channels come
  from a pool which you created by specifying a channel count in
  System::init. Channel handles get re-used if old sounds have stopped on
  them. If all channels are playing, then one of the existing channels will get
  stolen based on the lowest priority sound. Make sure this doesn't happen by
  simply increasing the channel count in System::init.
- **A channel becomes invalid once it is finished playing.** This means you
  can't update it, and doing so would be pointless anyway because it isn't
  going to start again. Referencing a stopped channel will most likely result
  in an FMOD_ERR_INVALID_HANDLE.
Update. (This is important!)

It is important that System::update be called once per frame. Do not call this more than once per frame, as this is not necessary and is just inefficient. This function updates the following aspects of FMOD Ex.

- **Platform specific routines** such as the once a frame command packet send to the IOP on the PlayStation 2. Without the update no sound would be audible on this platform.
- **Virtual voice emulation.** Without update being called, virtual voices would pause.
- **3D voice calculation.** If update is not called, sounds will not audibly move in 3D even though the channel or listener has been had its 3D attributes set.
- **Geometry engine.** The FMOD polygon/geometry engine is updated from this function. Without it, the occlusion/obstruction properties defined by the user will not be audible.
- **Non realtime output.** FMOD_OUTPUTTYPE_NOSOUND_NRT and FMOD_OUTPUTTYPE_WAVWRITER_NRT need this function to be called to update to the output. (ie write to the file in FMOD_OUTPUTTYPE_WAVWRITER_NRT).
- **Streaming engine,** if FMOD_INIT_STREAM_FROM_UPDATE is specified. If the user has decided to drive the streaming engine themselves from the main thread, then update must be called regularly or the streamer will stutter and cause buffer underrun.
• **Shutdown.**

Call **System::release** to close the output device and free all memory associated with that object. Channels are stopped, but sounds are *not* released. You will have to free them first. You do not have to stop channels yourself. You can of course do it if you want, it is just redundant, but releasing sounds is good programming practice anyway. You do not have to call **System::close** if you are releasing the system object. **System::release** internally calls **System::close** anyway.
Resource usage configuration.

In application development, some developers will want to have all disk or memory access going through their own functions rather than using the default system. In FMOD Ex, you can configure the FMOD file system to use your own file routines with `System::setFileSystem`.

To make FMOD use your memory system, or to confine FMOD to 1 block of memory that it will not allocate outside of, use `Memory_Initialize`.
Firelight Technologies FMOD Ex
TRANSITIONING BETWEEN FMOD 3 AND FMOD EX. API DIFFERENCES

Introduction

This section will describe some of the differences between FMOD 3 and FMOD Ex, if you are used to the old API and have difficulty understanding the difference between the 2 APIs. It will answer some of the more common questions usually beginning with "What happened to.."
FMOD 3 had streams, sample and music APIs, now what?

All combined into the one class **Sound**. This leads to a much leaner and streamlined API. To create a stream just use `System::createStream` or FMOD_CREATESTREAM flag with `System::createSound`. Music files loaded with the old music api would just be opened as a stream. As you could now load these types of sounds as a static sample (yes you can decode a whole mod into memory as PCM) it would possibly take hundreds of megabytes of ram, so even if you specify `System::createSound` to load a mod/s3m/xm/it file, it will still open it as a stream. To force it to a sample (not used as often) simply use FMOD_CREATESSAMPLE flag in `System::createSound`. The old 'music' formats (mod/s3m/xm/it/midi) now being streams means you can also do cool things like place effects on music formats (`Channel::addDSP`), or treat them like a normal channel with `Channel::setVolume` / `Channel::setFrequency` etc and therefore can even 3d position them!
Channels are now objects instead of just integer handles

FMOD Ex now takes a more object oriented approach than FMOD 3. Channel objects are still reference counted though! So if the channel handle you have is stolen, FMOD Ex will still know not to update the newly playing channel with commands issued from the old channel handle. Channel stealing should be less prevalent now thanks to virtual channels. You can now allocate a pool of many hundreds or even thousands of channels which will never run out, and they all succeed when you try to play them all at once. This is thanks to FMOD Ex's new **virtual voice system**.
Volume and pan

FMOD 3 used volume 0-255 (silent to full volume) and pan 0-255 (left to right), but now FMOD Ex takes a floating point number for each. FMOD Ex now uses **0.0 to 1.0** for volume (silent to full), and **-1.0 to +1.0** for pan (left to right, 0.0 = center).
**Frequency**

FMOD 3 used integer frequencies. FMOD Ex now uses floating point frequencies. Now you can get far greater accuracy for sound playback (ie you can now set 44100.5 instead of having to choose between 44100 and 44101) which is important when trying to do exact playback synchronization between 2 streams of different bpm for example.
FSOUND_GetError is gone

FMOD 3 used a global error code for determining what an error was. This was a pretty bad design choice, as internal and multithreaded FMOD calls could contaminate the global error code.
FMOD Ex now uses a much cleaner error return code for every single function. This is not affected by the previously mentioned issues.
What happened to FSOUND_SetSFXMasterVolume? or How do I perform master volume?

FMOD 3 used this function to scale all non music oriented channel volumes. FMOD Ex now uses 'ChannelGroups' which are far more powerful, and to scale all channels by a master volume, just use `System::getMasterChannelGroup` then `ChannelGroup::setVolume`.

Using ChannelGroups you can now have multiple master volume groups, and other exciting features such as DSP submixing.
MOD/S3M/XM/IT channels used to take up channels in FMOD 3's main channel pool so I had to adjust FSOUND_Init, do I have to do this with System::init?

No. MOD/S3M/XM/IT (and now MIDI) have their own channel pools that do not affect the number in System::init. Just select a number of channels that YOU are going to use, don't worry about what FMOD Ex is doing internally. To you playing a MOD/S3M/XM/IT/MID uses 1 channel.
Where is `FSOUND_SetHWND`?

FMOD Ex is global focus, or windowless by default. If you really need to focus the audio on a particular window in FMOD_OUTPUTTYPE_DSOUND mode on win32/win64, pass the hwnd as the extradriverdata parameter in `System::init`
Where is FSOUND_GetCurrentLevels?

Use System::getWaveData or Channel::getWaveData. It is far more flexible.
FSOUND_Update is now System::update.

Call System::update once a frame in your game loop. This is necessary to update various aspects of FMOD Ex.
Firelight Technologies FMOD Ex
CHANNEL MANAGEMENT AND VIRTUAL VOICES.

Introduction

FMOD Ex now includes an efficient virtual voice management system. This tutorial will explain how it works and what is the advantage of using virtual voices.
What are virtual voices?

What is a virtual voice and how is this different to a hardware or software voice?

Notice this screenshot of the FMOD Ex virtual voices example. It is playing 50 sounds at once, but only 10 are audible.

On limited sound hardware, which generally only has 32 to 64 voices, it can be challenging to manage your whole game's audio voice allocation when you want to have hundreds or even thousands of sounds playing at once in world (for example in a dungeon there might be 200 torches burning on walls in various places all playing a crackling burn noise).

FMOD Ex now allows the user to play as many sounds as they require, and will automatically allocate the limited number of hardware or software voices to the most important sounds to the listener.
This could mean in as in the above example, in a 3D world the 10 closest sounds are audible and the rest become 'virtual'.
Notice in the above screenshot red sounds are audible, and the blue sounds are inaudible and 'virtual'. According to the user though, there are actually 50 sounds playing at once.

'Virtual' voices are not allocated to a hardware or software voice, and are usually the least important sounds to the listener. These are 'emulated' voices. They will update their play cursors and seem to be playing like a normal sound, but will not be audible. As the user moves around the world, or a 'virtual' voice suddenly becomes more important than one that is actually audible, FMOD's virtual voice manager will swap the two voices, and the sound that was previously virtual will now become audible at its correct position in time.

A voice can be queried if it is virtual or not by using the `Channel::isVirtual` function. This is usually only for informational purposes.
What if some sounds are more important than others?

First we will take the case of 2D sounds that are all playing at the same volume. How do you make sure one sound stays audible and the others possibly become virtual if too many sounds are playing? The answer is to use the `Channel::setPriority` or `Sound::setDefaults` function.

By making one sound have a higher priority than another, it will be given priority to be audible while its competitor will be swapped out and become virtual.

For example if there were 10 sounds playing and only 10 real voices, and an 11th voice wants to be played. If the new sound has a higher priority than the voices playing, it will be played as audible and one of the original 10 will become virtual, because the new sound is more important.

Important sounds should have higher priority and it is up to the user to decide if some sounds should be more important than others. An example of an important sound might be a 2D menu or GUI sound or beep that needs to be heard above all other sounds.

Volume of a sound is a secondary determining factor between sounds of equal priority. If a group of sounds have the same priority, the loudest sound will be the most important. In a 3d world this usually means the closest sounds will be more important and the further away sounds, or the quieter sounds will be less important and will possibly become virtual.
What if I run out of virtual voices?

If you try to play more sounds than there are virtual voices, then FMOD Ex channel manager will try to find the least important sound and replace it with the new sound. This means the channel that has been replaced will stop and become invalid.

If a channel handle that has been kicked out by a new channel becomes invalid, any commands that are used on that channel handle will return **FMOD_ERR_INVALID_HANDLE**.
How do I set the number of real voices and virtual voices?

To set the number of virtual voices FMOD Ex will use, call `System::init` with the number of virtual voices specified in the maxchannels parameter.

For hardware voices, generally you don't set the number of these available on a sound device, such as on a console or sound card. Usually you are provided with a number of hardware channels to use. For example, PlayStation 2 always has 48 hardware voices.

On a sound card, this is variable depending on the manufacturer. You can find out the number of available hardware channels with `System::getHardwareChannels`.

If you want to limit the number of hardware channels below its capacity, you can use `System::setHardwareChannels`.

This type of voice is used if the sound is created with `FMOD_HARDWARE` flag.

To set the number of software mixed channels available, use `System::setSoftwareChannels`. You can set this to 0 if you don't want any software mixed voices.

This type of voice is used if the sound is created with `FMODSOFTWARE` flag.
How many virtual voices should I set?

How many sounds are you trying to play at once without losing control of the channel handles?
This figure is up to you, but remember that more channels = more CPU and memory usage.

If you have 32 real hardware or software channels available to you and don't want to play more than this at once, then you might only need 32 virtual voices. This will mean a 1 to 1 relationship between real voices and virtual voices and sounds will never become emulated and be swapped out. Instead if you play more than the specified amount of channels, it will 'kick out' other lower priority channels.

If you have 32 real hardware or software channels available and you want to be able to safely play 100 at once, or 1000 at once, then set it to 100 or 1000 at once. Figures around the 1000 mark playing at once might start to show non negligible amounts of CPU and memory usage so be wary of this. Use System::getCPUUsage and FMOD::Memory_GetStats to determine this.
Can I make silent sounds go virtual?

Yes. To do this enable the `FMOD_INIT_VOL0_BECOMES_VIRTUAL` flag in `System::init`.

To configure this even further, you can change it from volume 0, to a higher volume, between 0 and 1. For example if you set the level for voices to go virtual at 0.1, everything below this audibility would go virtual. Warning if this is set too high, sounds may appear to 'cut out' before they are silent. Use `System::setAdvancedSettings`, and the 'vol0virtualvol' member of `FMOD_ADVANCEDSETTINGS`. 
How do I tell if a Channel is virtual or not?

See Channel::isVirtual
What if I don't like the sound of a voice going from virtual to real and playing half way through the sound, or near the end of the sound? (sounds like a bug!)

You can either use Sound or Channel priorities to stop it going virtual in the first place, or you have the option to have a voice start a from the beginning instead of half way through, by using the `FMOD_VIRTUAL_PLAYFROMSTART` flag with `System::createSound`, `System::createStream`, `Sound::setMode` or `Channel::setMode`.

As described above, only the quietest, least important sounds should be swapping in and out, so you shouldn't notice sounds 'swapping in', but if you have a low number of real voices, and they are all loud, then this behaviour could become more noticable and may sound bad.

Another option is to simply call `Channel::isVirtual` and stop the sound, but don't do this until after a `System::update`! After playsound, the virtual voice sorting needs to be done in `System::update` to process what is really virtual and what isn't.
Firelight Technologies FMOD Ex
3D SOUND

Introduction.

This section will introduce you to using 3D sound with FMOD Ex. With it you can easily implement interactive 3D audio and have access to features such as 5.1 or 7.1 speaker output, and automatic attenuation, doppler and more advanced psychoacoustic 3D audio techniques.
Loading sounds as '3D'.

When loading a sound or sound bank, the sound must be created with `System::createSound` or `System::createStream` using the `FMOD_3D` flag.

```cpp
result = system->createSound("../media/drumloop.wav", FMOD_3D, 0
if (result != FMOD_OK)
{
    HandleError(result);
}
```

This will try and allocate a sound using hardware mixing by default. If there is no hardware mixing available, it will use software mixing as fallback. To specifically load a sound in hardware or software simply add `FMOD_HARDWARE` or `FMOD_SOFTWARE`

```cpp
result = system->createSound("../media/drumloop.wav", (FMOD_MODE)(FMOD_HARDWARE | FMOD_3D), 0,
if (result != FMOD_OK)
{
    HandleError(result);
}
```

Note that once the sound is loaded, on Win32 and `FMOD_OUTPUTTYPE_DSOUND` output (the default on Win32), you can't change the mode from `FMOD_3D` to `FMOD_2D` and vice versa. This is a limitation of DirectSound.

Using `FMOD_SOFTWARE` instead of `FMOD_HARDWARE` alleviates this issue, and other platforms that support hardware (ie Xbox, PS2, Gamecube) allow switching between 2D and 3D.

It is generally best not to try and switch between 3D and 2D at all, if you want though, you can change the sound or channel's mode to `FMOD_3D_HEADRELATIVE` at runtime which places the sound always relative to the listener, effectively sounding 2D as it will always follow the listener as the listener moves around.
Distance models and linear rolloff vs logarithmic.

**Logarithmic**

This is the default FMOD 3D distance model. All sounds naturally attenuate (fade out) in the real world using a logarithmic attenuation. The flag to set to this mode is `FMOD_3D_LOGROLLOFF` but if you're loading a sound you don't need to set this because it is the default. It is more for the purpose or resetting the mode back to the original if you set it to `FMOD_3D_LINEARROLLOFF` at some later stage.

When FMOD uses this model, *mindistance* of a sound / channel, is the distance that the sound starts to attenuate from. This can simulate the sound being smaller or larger. By default, for every doubling of this mindistance, the sound volume will halve. This rolloff rate can be changed with `System::set3DSettings`.

As an example of relative sound sizes, we can compare a bee and a jumbo jet. At only a meter or 2 away from a bee we will probably not hear it any more. In contrast, a jet will be heard from hundreds of meters away. In this case we might set the bee's mindistance to 0.1 meters. After a few meters it should fall silent. The jumbo jet's mindistance could be set to 50 meters. This could take many hundreds of meters of distance between listener and sound before it falls silent. In this case we now have a more realistic representation of the loudness of the sound, even though each wave file has a fully normalized 16bit waveform within. (ie if you played them in 2D they would both be the same volume).

The *maxdistance* does not affect the rate of rolloff, it simply means the distance where the sound stops attenuating.

**Don't set the maxdistance** to a low number unless you want it to artificially stop attenuating. This is usually not wanted. Leave it at its default of 10000.0.

**Linear**

This is an alternative distance model that FMOD has introduced. It is supported by adding the `FMOD_3D_LINEARROLLOFF` flag to `System::createSound` or
Sound::setMode / Channel::setMode.
This is a more fake, but usually more game programmer friendly method of attenuation. It allows the 'mindistance' and 'maxdistance' settings to change the attenuation behaviour to fading linearly between the two distances. Effectively the mindistance is the same as the logarithmic method (ie the minimum distance before the sound starts to attenuate, otherwise it is full volume), but the maxdistance now becomes the point where the volume = 0 due to 3D distance.
The attenuation inbetween those 2 points is linear.
**Some global 3D settings.**

The 3 main configurable settings in FMOD Ex that affect all 3D sounds are:

- Doppler factor. This is just a way to exaggerate or minimize the doppler effect.
- Distance factor. This allows the user to set FMOD to use units that match their own (ie centimeters, meters, feet)
- Rolloff scale. Affects 3d sounds that use FMOD_3D_LOGROLLOFF. Controls how fast all sounds attenuate using this mode.

All 3 settings can be set with `System::set3DSettings`. Generally the user will not want to set these.
• **Velocity and keeping it frame rate independent.**

Velocity is only required if you want doppler effects. Otherwise you can pass 0 or NULL to both `System::set3DListenerAttributes` and `Channel::set3DAttributes` for the velocity parameter, and no doppler effect will be heard.

This must be stressed again. It is important that the velocity passed to FMOD Ex is meters **per second** and not meters **per frame**. Notice the difference.

To get the correct velocity vector, use vectors from physics code etc, and don't just subtract last frames position from the current position. This is affected by framerate. The higher the framerate the smaller the position deltas, and therefore smaller doppler effects, which is incorrect.

If the only way you can get the velocity is to subtract this and last frame's position vectors, then remember to time adjust them from meters per frame back up to meters per second.

This is done simply by scaling the difference vector obtained by subtracting the 2 position vectors, by one over the frame time delta.

Here is an example.

```
velx = (posx-lastposx) * 1000 / timedelta;
velz = (posy-lastposy) * 1000 / timedelta;
velz = (posz-lastposz) * 1000 / timedelta;
```

timedelta is the time since the last frame in milliseconds. This can be obtained with functions such as `timeGetTime()`.

So at 60fps, the timedelta would be 16.67ms. if the source moved 0.1 meters in this time, the actual velocity in meters per second would be:

```
vel = 0.1 * 1000 / 16.67 = 6 meters per second.
```

Similarly, if we only have half the framerate of 30fps, then subtracting position deltas will gives us twice the distance that it would at 60fps (so it would have moved 0.2 meters this time).

```
vel = 0.2 * 1000 / 33.33 = 6 meters per second.
```
- **Orientation and left-handed vs right-handed coordinate systems.**

Getting the correct orientation set up is essential if you want the source to move around you in 3D space.

FMOD Uses a left handed coordinate system by default, (+X = right, +Y = up, +Z = forwards), which is the same as DirectSound3D and A3D.

If you use a different coordinate system, then you will need to flip certain axis or even swap them around inside the call to System::set3DListenerAttributes and Channel::set3DAttributes.

Take the right handed coordinate system, where +X = right, +Y = up, +Z = backwards or towards you. To convert this to FMOD coordinate system simply negate all instances of the Z coordinate for listener and sound position and velocity, as well as listener up and forward vector Z components.

To make things easier for people using the right handed coordinate system, you can initialize FMOD Ex using FMOD_INIT_3D_RIGHTHANDED in System::init and not do any conversion. FMOD will automatically convert its internal 3D calculations to be right handed instead of left handed.
• A typical game loop.

3D sound and the FMOD channel management system need to be updated once per frame.
To do this use System::update
This would be a typical example of a game audio loop.

do
{
    UpdateGame(); // here the game is updated and the sources
    system->set3DListenerAttributes(0, &listener::_pos, &listener::_ve
    system->update(); // needed to update 3d engine, once per fram
} while (gamerunning);

Most games usually take the position, velocity and orientation from the camera's vectors and matrix.
• Stereo and multichannel sounds can be 3D!

A stereo sound when played as 3d, will be split into 2 mono voices internally which are separately 3d positionable. Multi-channel sounds are also supported, so an 8 channel sound for example will allocate 8 mono voices internally in FMOD.
To rotate the left and right part of the stereo 3d sound in 3D space, use the \texttt{Channel::set3DSpread} function.
By default the subchannels position themselves in the same place, therefore sounding 'mono'.
- **Split screen / multiple listeners.**

In some games, there may be a split screen mode. When it comes to audio, this means that FMOD Ex has to know about having more than 1 listener on the screen at once.

This is easily handled via `System::set3DNumListeners` and `System::set3DListenerAttributes`.

If you have 2 player split screen, then for each 'camera' or 'listener' simply call `System::set3DListenerAttributes` with 0 as the listener number of the first camera, and 1 for the listener number of the second camera. `System::set3DNumListeners` would be set to 2.

That's all there is to it. You may notice an audible difference, because fmod does a few things to avoid confusion with the same sound being viewed from different viewpoints.

- 1. It turns off all doppler. This is because one listener might be going towards the sound, and another listener might be going away from the sound. To avoid confusion, the doppler is simply turned off.
- 2. All audio is mono. If to one listener the sound should be coming out of the left speaker, and to another listener it should be coming out of the right speaker, there will be a conflict, and more confusion, so all sounds are simply panned to the middle. This removes confusion.
- 3. Each sound is played only once as it would with a single player game, saving voice and cpu resources. This means the sound's effective audibility is determined by the closest listener to the sound. This makes sense as the sound should be the loudest to the nearest listener. Any listeners that are further away wouldn't have any impact on the volume at this point.
• **Speaker modes / output.**

To get 5.1 sound is easy. If the sound card supports it, then any sound using `FMOD_3D` and `FMOD_HARDWARE` will automatically position itself in a surround speaker system, and only the user has to be sure that the speaker settings in the operating system are correct so that the sound device can output the audio in 5.1 or 7.1.

**You do not need to call** `System::setSpeakerMode!`. This function is only used to configure FMOD Ex's software mixing engine. See the next paragraph on this.

For sounds created with `FMOD_SOFTWARE`, by default sound is emulated through a simple stereo output. This involves panning and volume attenuation. To enable FMOD software mixing to use 5.1 output, you can use `System::setSpeakerMode`. But note! This function increases the CPU mixing burden slightly as it now has to software mix into a 6 or 8 channel buffer instead of a stereo buffer.
Firelight Technologies FMOD Ex
3D Reverb Tutorial

Introduction

It is common for environments to exhibit different reverberation characteristics in different locations. Ideally as the listener moves throughout the virtual environment, the sound of the reverberation should change accordingly. This change in reverberation properties can be modeled in FMOD Ex by using the built in 3D Reverb API. Note: This technique applies to software channels.
**3D Reverbs**

The 3D reverb system works by allowing you to place multiple virtual reverbs within the 3D world. Each reverb defines:

- Its position within the 3D world
- The area, or sphere of influence affected by the reverb (with minimum and maximum distances)
- The reverberation properties of the area

At runtime, FMOD Ex interpolates (or morphs) between the characteristics of 3D reverbs according to the listener's proximity and the position and overlap of the reverbs. This method allows FMOD Ex to use a single reverb DSP unit to provide a dynamic reverberation within the 3D world. This process is illustrated in the image below.

When the listener is within the sphere of effect of one or more 3D reverbs, the
listener hears a weighted combination of the affecting reverb. When the listener is outside the coverage of all 3D reverb, the reverb reverts to the default ambient reverb setting. It is important to note that this technique will only place add reverb to 3D sounds. To place reverb effects on 2D sounds, you will need to make an additional to call System::setReverbProperties (which will incur a small CPU and memory hit).

The interpolation of 3D reverb is only an estimation of how the multiple reverberations within the environment may sound. In some cases, greater realism is required. In these situations we suggest using multiple physical reverbs as described in the tutorial 'Using multiple reverb'.
Set the default ambient reverb

In this section will we look at setting the default ambient reverb settings. This is important, as FMOD Ex will use these settings when the listener is not standing within an area affected by any of the 3D reverbs. In this example, we will use the reverb present `FMOD_PRESET_OFF`.

```c
FMOD_REVERB_PROPERTIES prop1 = FMOD_PRESET_OFF;
system->setReverbAmbientProperties(&prop1);
```
Create a 3D Reverb

We will now create a virtual reverb, using the call `System::createReverb`, then set the characteristics of the reverb using `Reverb::setProperties`.

```cpp
FMOD::Reverb *reverb;
result = system->createReverb(&reverb);
FMOD_REVERB_PROPERTIES prop2 = FMOD_PRESET_CONCERTHALL;
reverb->setProperties(&prop2);
```
Set 3D Attributes

The 3D attributes of the reverb must now be set. The method `Reverb::set3DAttributes` allows us to set the origin position, as well as the area of coverage using the minimum distance and maximum distance.

```c
FMOD_VECTOR pos = { -10.0f, 0.0f, 0.0f };  
float mindist = 10.0f;  
float maxdist = 20.0f;  
reverb->set3DAttributes(&pos, mindist, maxdist);
```

As the 3D reverb uses the position of the listener in its weighting calculation, we also need to ensure that the location of the listener is set using `System::set3dListenerAttributes`.

```c
FMOD_VECTOR listenerpos = { 0.0f, 0.0f, -1.0f };  
system->set3DListenerAttributes(0, &listenerpos, 0, 0, 0);
```
Firelight Technologies FMOD Ex
FMOD_NONBLOCKING flag and asynchronously loading data

Introduction

The `FMOD_NONBLOCKING` flag is used so that sounds can be loaded without affecting the framerate of the application. Normally, loading operations can take a large or significant amount of time, but with this feature, sounds can be loaded in the background without the application skipping a beat.
Creating the sound.

Simply create the sound as you normally would but add the **FMOD_NONBLOCKING** flag.

```cpp
FMOD::Sound *sound;
result = system->createStream("../media/wave.mp3", FMOD_NONBLOCKING,
ERRCHECK(result);
```

Now the sound will open in the background, and you will get a handle to the sound immediately. You cannot do anything with this sound handle except call `Sound::getOpenState`. Any other attempts to use this sound handle will result in the function returning **FMOD_ERR_NOTREADY**.
Getting a callback when the sound loads.

When the sound loads or the stream opens, you can specify a callback using the nonblockcallback member of the `FMOD_CREATESOUNDEXINFO` structure that is called when the operation is completed. Firstly the callback definition.

```c
FMOD_RESULT F_CALLBACK nonblockcallback(FMOD_SOUND *sound, FMOD_RESULT result) {
    FMOD::Sound *snd = (FMOD::Sound *)sound;
    printf("Sound loaded! (%d) %s\n", result, FMOD_ErrorString(result));
    return FMOD_OK;
}
```

And then the createSound call.

```c
FMOD_RESULT result;
FMOD::Sound *sound;
FMOD_CREATESOUNDEXINFO exinfo;

memset(&exinfo;, 0, sizeof(FMOD_CREATESOUNDEXINFO));
exinfo.cbsize = sizeof(FMOD_CREATESOUNDEXINFO);
exinfo.nonblockcallback = nonblockcallback;

result = system->createStream("./media/wave.mp3", FMOD_NONBLOCKING, ERRCHECK(result));
```
Waiting for the sound to be ready and using it.

As mentioned, you will have to call Sound::getOpenState to wait for the sound to load in the background. You could do this, or just continually try to call the function you want to call (ie System::playSound) until it succeeds. Here is an example of polling the sound until it is ready, then playing it.

```cpp
FMOD_RESULT result;
FMOD::Sound *sound;
result = system->createStream("../media/wave.mp3", FMOD_NONBLOCKING, ERRCHECK(result));

do {
    FMOD_OPENSTATE state;
    result = tmpsnd->getOpenState(&state, 0, 0);
    ERRCHECK(result);

    if (state == FMOD_OPENSTATE_READY && !channel)
    {
        result = system->playSound(FMOD_CHANNEL_FREE, sound,
                                    ERRCHECK(result));
    }

    GameCode();
} while (1)
```

The second loop will simply retry playsound until it succeeds.
Creating the sound as a streamed FSB file.

An FSB file will have subsounds in it, so if you open it as a stream, you may not want FMOD seeking to the first subsound and wasting time. You can use the initialsubsound member of the `FMOD_CREATESOUNDEXINFO` structure to make the non-blocking open seek to the subsound of your choice.

```c
FMOD_RESULT result;
FMOD::Sound *sound;
FMOD_CREATESOUNDEXINFO exinfo;

memset(&exinfo; , 0, sizeof(FMOD_CREATESOUNDEXINFO));
exinfo.cbsize = sizeof(FMOD_CREATESOUNDEXINFO);
exinfo.initialsubsound = 1;

result = system->createStream("../media/sounds.fsb", FMOD_NONBLOCKING);
ERRCHECK(result);

Then get the subsound you wanted with `Sound::getSubSound`.
```
Getting a subsound.

`Sound::getSubSound` is a free function call normally, all it does is return a pointer to the subsound, whether it be a sample or a stream. It does not execute any special code besides this. What it would cause if it was a blocking stream though, is `System::playSound` stalling several milliseconds or more while it seeks and reflushes the stream buffer. Time taken can depend on the file format and media.

If the parent sound was opened using `FMOD_NONBLOCKING`, then it will set the `subsound` to be `FMOD_OPENSTATE_SEEKING` and it will become not ready again until the seek and stream buffer flush has completed. When the stream is ready and `System::playSound` is called, then the playsound will not stall and will execute immediately because the stream has been flushed.
Firelight Technologies FMOD Ex
MEMORY MANAGEMENT AND CONSERVATION TUTORIAL

Introduction

This section will give some pointers on how to use and save memory in FMOD Ex by describing things that may not be so obvious upon first looking at the API.
Using a fixed size memory pool.

To make FMOD stay inside a fixed size memory pool, and not do any external allocs, you can use the `FMOD::Memory::Initialize` function.

i.e.

```cpp
result = FMOD::Memory::Initialize(malloc(4*1024*1024), 4*1024*1024,
ERRCHECK(result);
```

**Note** that this uses malloc. On Xbox 360 and Xbox you must use a different operating system alloc such as XPhysicalAlloc otherwise FMOD may not behave correctly. See "Platform specific issues" tutorials for more information on this.

Note that this function allows you to specify your own callbacks for alloc and free. In this case the memory pool pointer and length must be NULL. The 2 features are mutually exclusive.
Lowering sound instance overhead.

The FMOD_LOWMEM flag is used for users wanting to shave some memory usage off of the sound class.
This flag removes memory allocation for certain features like the 'name' field which isn't used often in games. When this happens, Sound::getName will return "(null)".
More memory will be stripped from the sound class in future versions of FMOD Ex when this flag is used. Currently the 'name' field is the biggest user of memory in the sound class so this has been removed first.
Using compressed samples.

To trade CPU usage vs Memory, FMOD Ex has a feature to play ADPCM, XMA and MP2/MP3 data compressed, without needing to decompress it to PCM first. This can save a large amount of memory.

On XBox 360, using this for XMA files incurs next to no extra CPU usage, as the Xbox 360 XMA hardware decoder does the data decompression in realtime. To enable this use the `FMOD_CREATECOMPRESSEDSAMPLE` flag. If this flag is used for formats other than the ones specified above, it will be ignored.

With the exception of XMA on Xbox 360 and ADPCM on Xbox, if `FMOD_CREATECOMPRESSEDSAMPLE` is used with an FMOD_HARDWARE buffer it will generate an `FMOD_ERR_NEEDSSOFTWARE` error.

**Note!** If you use `FMOD_CREATECOMPRESSEDSAMPLE` there will be a 'one off' memory overhead to allocate the appropriate pool of codecs depending on the format being loaded. See the next section on how to control this pool.
Controlling memory usage with settings.

- **System::setSoftwareFormat** 'maxinputchannels' is default to 6 to allow up to 6 channel wav files to be played through FMOD's software engine. Setting this to a lower number will save memory across the board. If the highest channel count in a sound you are going to use is stereo, then set this to 2.
- For sounds created with **FMOD_CREATECOMPRESSESAMPLE**, **System::setAdvancedSettings** allows the user to reduce the number of simultaneous XMA/ADPCM or MPEG sounds played at once, to save memory. The defaults are specified in the documentation for this function. Lowering them will reduce memory. Note the pool of codecs for each codec type is only allocated when the first sound of that type is loaded. Reducing XMA to 0 when XMA is never used will not save any memory.
- For streams, setting **System::setStreamBufferSize** will control the memory usage for the stream buffer used by FMOD for each stream. Lowering the size in this function will reduce memory, but may also lead to stuttering streams. This is purely based on the type of media the FMOD streamer is reading from (ie CDROM is slower than harddisk), so it is to be experimented with based on this.
- Reducing the number of channels used will reduce memory. **System::init** and **System::setSoftwareChannels** give control over maximum number of virtual voices and software voices used. You will need to make sure you specify enough voices though to avoid channel stealing.
• Tracking FMOD memory usage.

Using `FMOD::Memory_GetStats` is a good way to track FMOD memory usage, and also find the highest amount of memory allocated at any time, so you can adjust the fix memory pool size for the next time.
Firelight Technologies FMOD Ex
The FMOD DSP Network

Introduction

This section will introduce you to the FMOD Ex advanced DSP system. With this system you can implement custom filters or complicated signal chains to create high quality and dynamic sounding audio. The FMOD Ex DSP system is an incredibly flexible mixing engine that goes far beyond FMOD 3's capabilities. Its emphasis on quality, flexibility and efficiency makes it an extremely powerful system when used to its full potential.

The figure below shows a representation of what a very basic FMOD DSP network looks like.

Audio data flows from the right to the left, until it finally arrives at the soundcard, fully mixed and processed.

The above image was taken using the FMOD Profiler tool. You can profile your own DSP network as long as you specify 'FMOD_INIT_ENABLE_PROFILE' when initializing the FMOD Ex engine. The tool is located in the /tools directory of the SDK.

FMOD Designer also provides a built in profiler. You don't need to load a project or even use the event system. You can just use the 'Profiler' tab within FMOD Designer as a standalone application.
Data flow and Signal chain

When FMOD plays a sound on a channel (using System::playSound), it creates a small sub-network consisting of a Channel DSP Head and a Wavetable Unit. When FMOD plays a DSP on a channel (System::playDSP), it creates a small sub-network consisting of a Channel DSP Head and a Resampler Unit. The DSP that was specified by the user is then attached to this as an input.

This section will describe the units in more detail, from the origin of the data through to the soundcard, from right to left.

- **Wavetable Unit**
  This unit reads raw PCM data from the sound buffer and resamples it to the same rate as the soundcard. A Wavetable Unit is only connected when the user calls System::playSound. Once resampled, the audio data is then processed (or flows) at the rate of the soundcard. This is 48khz by default.

- **Channel DSP Head**
  This unit provides a place for extra DSP effects to be connect to, between the Wavetable Unit (if System::playSound was used), or a user specified DSP unit (if System::playDSP was used), and the ChannelGroup Unit that it belongs to.

  By default all channels connect to the Master Channel Group and it is this point where the channel volume and pan gets applied. A Channel DSP Head unit incurs no CPU penalty. The data is simply passed straight to its outputs.

- **ChannelGroup DSP Heads**
  The Master ChannelGroup is the default target for Channel DSP heads, and is owned by the System object. When multiple Channel DSP Heads are connected to a channel group, they are mixed together.

  This is the case for any DSP unit with multiple inputs. Other channel groups may also be created by the user, which means channels may target them instead. This happens when the user calls Channel::setChannelGroup. Channelgroups are there for sub-mixing. Effects can be placed after this
point between it and the ChannelGroup Target Unit.

- **ChannelGroup Target Unit**
  This is the target DSP unit for all ChannelGroups created by the user (with `System::createChannelGroup`) and the System ChannelGroup.
## Code Examples

In this section we will look at some basic techniques that can be used to manipulate DSP network. We shall start with the most basic signal chain (as shown in the image below) and identify the changes that occur to the DSP network with the provided code.

### Connecting a DSP effect to a channel

In this example we shall add an effect to a sound by connecting a DSP effect unit to the channel. The code below starts by playing a sound, then creates a DSP unit with `System::createDSPByType` and adds it to the DSP network using `Channel::addDSP`.

```cpp
result = system->playSound(FMOD_CHANNEL_FREE, sound, false, &channel);
result = channel->setVolume(volume);
result = system->createDSPByType(FMOD_DSP_TYPE_ECHO, &dsp);
result = channel->addDSP(dsp, 0);
```

The figure below shows the FMOD Echo effect inserted into the DSP network between the WaveTable unit and FMOD Channel DSPHead.

### Connecting a DSP effect to a channel group

In this example we shall add an effect to a channel group. The advantage of this is multiple channels can be treated with a single DSP effect (therefore is more resource friendly than adding the same affect to every individual WaveTable unit). The code below gets the Master Channel Group and attached an echo effect to it.

```cpp
result = system->playSound(FMOD_CHANNEL_FREE, sound, false, &channel);
result = channel->setVolume(volume);
result = system->createDSPByType(FMOD_DSP_TYPE_ECHO, &dsp);
result = system->getMasterChannelGroup(&channelgroup);
```
result = channelgroup->addDSP(dsp, 0);

The figure below shows the FMOD Echo effect attached to the ChannelGroup: FMOD master group. Note: The ChannelGroup: FMOD master group on the right is used to submix all the channels entering the channel group.

### Connecting a DSP effect on the system

In this example we shall add an effect on the final point of the DSP network before the soundcard. This means all sounds within the network will be affected. This is done by creating a DSP with `System::createDSPByType` and adding it the DSP network using `System::addDSP`.

```cpp
result = system->playSound(FMOD_CHANNEL_FREE, sound, false, &channel);
result = channel->setVolume(volume);
result = system->createDSPByType(FMOD_DSP_TYPE_ECHO, &dsp);
result = system->addDSP(dsp, 0);
```

The figure below shows the FMOD Echo effect inserted into the DSP network between the ChannelGroup Target unit (the final system bus) and sound card.

### Creating a wet / path for a channel group

This example demonstrates a more complex scenario, in which we shall create a wet/dry path for a channel group. This is very similar to using a 'send effect' signal chain on a hardware mixer. The first step is to add a channel group, which we shall call 'my channelgroup'.

```cpp
result = system->playSound(FMOD_CHANNEL_FREE, sound, false, &channel);
result = channel->setVolume(volume);
result = system->createChannelGroup("my channelgroup", &channelgroup);
```

In the figure below we can see the new channel group 'ChannelGroup: my channelgroup' is now included in DSP network. It is important to note that this channel group runs separately from the master group, and does not pass through it.
Now we will add an echo DSP effect to 'my channelgroup'.

```c
result = system-&gt;createDSPByType(FMOD_DSP_TYPE_ECHO, &dsp;);
result = channelgroup-&gt;addDSP(dsp, 0);
```

This is reflected in the figure below.

In this step we shall create a new path from the original channel signal path to the path that includes the echo effect. To do this we must take the channel DSP unit head node and connect it to the head node of the 'my channelgroup' channel group using the methods `DSP::getInput` and `DSP::addInput`.

```c
FMOD::DSP *channeldsphead;
FMOD::DSP *channelgroupdsphead;
result = channel-&gt;getDSPHead(&c;hanneldsphead);
result = channelgroup-&gt;getDSPHead(&c;hannelgroupdsphead);
result = channelgroupdsphead-&gt;getInput(0, &c;hannelgroupdsphead, 0);
result = channelgroupdsphead-&gt;addInput(channeldsphead, 0);
```

You can see in the figure below there is now a connection from the channel DSP head to both the channel groups 'FMOD master group' (dry) and 'my channelgroup' (wet).

**Disable an effect**

To disable an effect simply use the setBypass method. The code below plays a sound, adds an effect then bypasses it.
Playing DSP Oscillators

Within FMOD Ex the DSP units can be used to generate signals. The following example generate as 440Hz sine wave (which is the default signal settings).

The figure below shows an Oscillator unit in the DSP network. The Resampler Unit is a generic buffered resampler that is usually used to connect Oscillator unit to its channel.
A look at a more complex DSP Network

In this section we will work through a more complex DSP Network. Use the icons and terms in **bold** to find the appropriate reference points within the example image.

**Submix** - see the icon 🎧. When multiple inputs converge into one unit, they are mixed together. This is called a 'submix'.

**FMOD Channel DSPHead Unit** - see the icon 🎧. Any time `System::playSound` or `System::playDSP` is called, an **FMOD Channel DSPHead Unit** is attached to the network. When it is stopped, it is disconnected from the network.

To the right of each DSPHead Unit, there is either an **FMOD WaveTable Unit**, an **FMOD DSP Codec** or an **FMOD Resampler Unit**.

- A WaveTable unit is a unit that plays standard PCM data (ie a standard wav file).
- A DSP Codec is a compressed realtime sample, created by `FMOD_CREATECOMPRESSEDSAMPLE` (in this case an mp3).
- Resampler Unit is a generic buffered resampler that is usually used when connecting generic DSP units to a channel (wavetable and DSPcodec usually have a built in resampler, standard DSP units do not so one has to be inserted if it is to be played on a channel, so `Channel::setFrequency` can work).

These things feed data into a **Channel DSPHead Unit**, then it feeds the data
onto its parent and so on.

The 3 channels that have an **FMOD WaveTable Unit**, are playing a pcm wave file, and have been connected to a channelgroup. **ChannelGroup: B**. This was done with **Channel::setChannelGroup**

The 3 channels that have an **FMOD DSP Codec** Unit, are playing an mp3 file and are connected to a channelgroup. **ChannelGroup A**. This was done with **Channel::setChannelGroup**

The channel that has **UNIT A** attached to it is not attached to a user created ChannelGroup. It is connected to the default system ChannelGroup called the "master channelgroup" which is the ChannelGroup all channels are played on if no other is specified. You can get a handle to this channelgroup with **System::getMasterChannelGroup**.

The **UNIT A** unit was created by the user with **System::createDSP** and was played with **System::playDSP**. It does nothing but act as a submix target, so the user has added 3 oscillators by creating them with **System::createDSPByType**, and adding them to **UNIT A** with **DSP::addInput**.

You may notice that **ChannelGroup: B** is also connected to **ChannelGroup: A** as an input. This means the result of the submix of **ChannelGroup: B** is fed into **ChannelGroup: A**. The connection of B to A was done with **ChannelGroup::addGroup**

ChannelGroup A and ChannelGroup B have 2 boxes each. This is because they both have a **DSP effect applied to them**. Without a DSP effect added, it just has one box, for optimization reasons (less memory, less cpu usage). The first box on the left is the head node, the second box on the right with the same name is the **mix target** for the channels. Inbetween is an **FMOD Echo DSP** (**FMOD_DSP_TYPE_ECHO**) effect on **ChannelGroup: B**, and a **FMOD Lowpass Simple** (**FMOD_DSP_TYPE_LOWPASS_SIMPLE**) effect on **ChannelGroup: A**. These were added with **ChannelGroup::addDSP**

Reverb has been enabled with **System::setReverbProperties** and can be seen as a DSP node called **SFX Reverb**. Notice all channel based DSP units have a connection going to it. This is the **wet path** - see the icon 🎧. The other paths go directly to the soundcard and bypass the SFX Reverb unit, therefore it is the **dry path** - see the icon 📺. To control the wet/dry mix you can use the
Channel::setReverbProperties. Internally this function just calls DSPConnection::setMix. Only channels are interested in this unit. Things like channelgroups and other units do not need to connect to this unit.

If there was no reverb enabled, the secondary links/outputs (on the FMOD WaveTable Unit/FMOD DSP Codec/FMOD Resampler Unit units) would be absent.

At the end of the mix, ChannelGroup: FMOD master group, ChannelGroup: A and SFX Reverb all get submixed into the FMOD ChannelGroup Target Unit

Finally the result of that submix gets sent to the FMOD SoundCard Unit which is the final destination.

If a DSP effect was to be added with System::addDSP, it would be inserted between FMOD SoundCard Unit and FMOD ChannelGroup Target Unit. You could see why in that case it would affect all sound in the DSP network.

If a DSP effect was to be added with Channel::addDSP, it would be inserted to the right of the FMOD Channel DSPHead Unit. If this happened you should be able to see why only the channel would be affected by the DSP effect.
Firelight Technologies FMOD Ex
Introduction.

This section provides more technical information on how to use the FMOD designer API, and how resource allocation is handled to allow the programmer to account for performance and memory issues. Just to provide some background information, the whole FMOD designer API sits on top of the low level FMOD API. This means it contains an FMOD::System object and uses all of the low level functions of the FMOD api to achieve its functionality.
Files should you receive from the sound designer.

When the sound designer provides you with a project, they must provide you with the following files.

- 1 `.FEV` file. An FEV file is the compiled sound designer project which you will load with `EventSystem::load`.
- 1 or more `.FSB` files. These files are raw audio data. They do not contain any event or sound designer data.
- Optionally, a project report `project name.txt`. This is a file that describes the events to the programmer and any associated notes, along with the parameters for each event and their min/max values.
- Other files are working files (such as `.cache`), do not ship these.
The programmer must work with the sound designer to organize banks and event groups to conserve memory!

Event groups should be used to control loading strategies, they are not just for aesthetic purposes, they are for loading purposes. The branches of an event tree are what you use to load when in the game code. See the "Event tree group strategies and loading / memory allocation issues" section below for very important issues related to loading and memory usage.
• Creating and initializing the EventSystem object.

Here is a typical bit of initialization code that you would call at the start of your project.

```cpp
FMOD::EventSystem *eventsystem = 0;
result = FMOD::EventSystem_Create(&eventsystem);
ERRCHECK(result);
result = eventsystem->init(256, FMOD_INIT_NORMAL, 0);
ERRCHECK(result);
```

If you want to configure the lower level FMOD engine before initializing the event system (ie select sound card driver, set speaker mode etc), then call `EventSystem::getSystemObject`.

```cpp
FMOD::EventSystem *eventsystem = 0;
FMOD::System *system = 0;
result = FMOD::EventSystem_Create(&eventsystem);
ERRCHECK(result);
result = eventsystem->getSystemObject(&system);
ERRCHECK(result);
(..Use System API here from fmod.hpp.)
result = eventsystem->init(256, FMOD_INIT_NORMAL, 0);
ERRCHECK(result);
```

**Do not create your own FMOD::System object** using `FMOD::System_Create`.

This will cause 2 system objects to be active (the one you just created plus the one within the EventSystem), which means it will try to open the sound device twice. It also means 2 software mixers would be spawned. This is A Bad Thing.

Also do not try to create multiple EventSystem objects. If you want to load multiple projects, simply load them from the one EventSystem.
Important memory management issue for consoles (Xbox, Xbox 360, PlayStation2, GameCube, PlayaStation Portable, PlayStation 3):

Memory management is a consideration that must be taken note of. On certain machines the default memory allocation is inefficient (ie the page size is way too big on Xbox 360 meaning megabytes of lost memory), or certain FMOD features just wont work without a memory pool (ie Xbox must have one single block of contiguous memory for audio buffers, and on Xbox 360 XMA buffers MUST reside within memory allocated with XPhysicalAlloc, otherwise the system will crash).

Before calling any fmod functions, first allocate a block of memory and pass it to FMOD. From then on it will not allocate any more memory itself.

\[
\text{result} = \text{FMOD}::\text{Memory}\_\text{Initialize}(\text{memblock}, \text{MEMSIZE}, 0, 0, 0);
\]
\[
\text{ERRCHECK} (\text{result});
\]

On Xbox 360 use XPhysicalAlloc.

Note that is the memory FMOD uses for all audio data including wave data, unless it is a machine with dedicated sound ram such as PlayStation 2 and GameCube.
In general it is usually a good idea to start off with a large memory block, then use \text{FMOD}::\text{Memory}\_\text{GetStats} to find out the maximum memory usage by FMOD during the progress of the game. \text{FMOD\_ERR\_MEMORY} will be returned from FMOD functions if it runs out of memory.

Virtual Voices.
The value you pass to \text{EventSystem::Init} for the number of 'channels' should be a high number resembling the highest number of voices you want to have playing at once (not audible! there is a difference).

The hardware may only have 32 hardware voices, but this number can be 64, 128, 256 or 1000, because FMOD has a \text{Virtual Voice System}. If you do not give this a high number, then voice stealing will take effect, and voices will drop out seemingly at random.
That is because new, more important sounds will be played, kicking out older
voices.

With a high enough number of virtual voices, no voices will be stolen, and FMOD will automatically swap voices in and out based on distance and priority. See the virtual voice tutorial in the tutorial for more information on this.
• **Load the project.**

Load the FEV file with `EventSystem::load`. This only allocates memory for the event tree structure. It does not allocate memory for sample data / wave bank data or even the memory for the event instances, which are the things you use to play and control the events later in the code. At this point the memory usage should be low and the memory allocated is for the low level software mixing engine and low level channel structures etc. You can load multiple FEV files with this function into `EventProjects`.
Traversing the event tree and getting events.

To traverse a tree you start at the root by calling `EventProject::getGroup` or `EventProject::getGroupByIndex` by specifying an event group at the root of the tree (ie 'examples' in this case).
From there you can call `EventGroup::getGroup` / `EventGroup::getGroupByIndex` to enter subgroups within the tree.
A full path can be entered within the GetGroup function, so that you don't have to manually traverse it yourself one at a time.
Finally you get a handle to an event with `EventGroup::getEvent` / `EventGroup::getEventByIndex`. 
These functions have loading (disk access) and memory allocation issues, which need to be considered in the next section.
• Event tree group strategies and loading / memory allocation issues. (Important!)

**Terminology.**

**event** - The leaf node in the tree. The thing you will obtain a handle to so that you can play it (and update its parameters).

**event group** - 'Folders' that contain events and other event groups. These are used for organizational and loading purposes.

**wave banks** - The .FSB files to be loaded. When memory allocation occurs for these, it means allocation for the raw PCM or compressed audio data.

**event instance memory** - Memory required to play the event(s). If an event has a 'max playbacks' value set in the designer tool, FMOD will allocate memory for that many instances, so that they can play simultaneously. Generally the memory footprint for this is small unless the sound designer has specified memory intensive DSP effects such as reverb, echo, chorus or flange. Other types of DSP effects generally do not allocate any memory (IIR effects such as lowpass filter, distortion).

**Organization of event hierarchy and banks.**

The event tree should be set up by grouping events into logical groups that will be loaded or used together, for example levels in a game, and common data. The reason for this is that you can load an entire branch's audio data with EventGroup::loadEventData. This data in particular is the (usually) larger wave bank data. This means it will load the waves from the FSB files referenced by the events in the tree. If it has already loaded sounds from the same FSB referenced from another event group, it will of course not try to re-load them.

**Note:** Sounds are either loaded selectively from an FSB, meaning you can pretty much put every sound into 1 wavebank, or you, as a programmer may prefer to preload a whole wavebank at once (ie EventSystem::preloadFSB) which means you may want to split waves into logical wavebank groups.

**Loading / Allocation overview:**

• EventSystem::load loads the FEV file, and only allocates a small amount of memory to hold the event tree structure.
• **EventGroup::loadEventData** loads all of the waves from the FSB files, necessary for the **specified group and its subgroups**. This function is recursive and traverses all subgroups.

  If you do not call **EventGroup::loadEventData**, FMOD loads the event's wave data, when it needs to when you call **EventGroup::getEvent** / **EventGroup::getEventByIndex**. If you do call **EventGroup::loadEventData** this won't happen. Calling the getevent function without loading the data first will mean a stall occurs as it loads. This is usually undesirable.

• **EventSystem::getGroup** / **EventGroup::getGroup** / **EventGroup::getGroupByIndex** allocate the **event instance memory** (including any DSP effect allocations) for the events in that group only, **EVENT_CACHEEVENTS** flag is used. This function is not recursive and does NOT traverse into subgroups.

  If **cacheevents = false** when getting a group, then FMOD will simply allocate the memory for the **event** when you try to get it. This means **EventGroup::getEvent** / **EventGroup::getEventByIndex** does the allocation.

• **EventGroup::freeEventData** unloads any wave data AND frees event instance memory for that group and all groups below it. This function is recursive and **DOES** traverse into subgroups.

• **EventGroup::getEvent** / **EventGroup::getEventByIndex** will not do any disk access if **EventGroup::loadEventData** was called, or allocate any memory if you have precached it with a **GetGroup** function with **cacheevents = true**.

**Just remember these things:**

1. Load your data at the loading phase of the game, with **EventGroup::loadEventData** from the root of a tree, generally for **static banks**.
2. Be selective when it comes to events using **streaming banks** that are in groups. If you only wanted 1 stream to play out of 10 in a group (ie all the tracks in your music group), then don't call **EventGroup::loadEventData** on the group, just call **EventGroup::getEvent** / **EventGroup::getEventByIndex** without having called **EventGroup::loadEventData** on the group, or using **cacheevents = true** on a **GetGroup** function.
3. **EventGroup::freeEventData** frees all memory related to a group and its children, including **wave bank** data and the **event instance memory**. If you call this then it will have to re-load and reallocate the data if you try to use that group again.
- **FMOD event system CPU usage.**

Now that the programmer has less control over the content of audio in the title, it may be easier to accidentally use more CPU than desired. It is generally easy to find out FMOD cpu usage by using the low level functionality of FMOD with `EventSystem::getSystemObject` and `System::getCPUUsage`. If the sound designer is using too many DSP effects, then the 'dsp' value will be high.

By default FMOD Designer puts all sounds in hardware, unless there is a DSP effect applied to the event. If there is then FMOD will load it into main ram, and mix the event on the CPU instead of in the audio chip.

Note all FMOD DSP effects are gradually being optimized using whatever SIMD capabilities are available on the machine. If an effect seems slower than it should be, it may possibly not be optimized, and request through support@fmod.org to have it done. There are 12 effects and 11 platforms to optimize, which means 132 routines have to be optimized, and they also have to be optimized for mono/stereo and multichannel purposes so that is 396 loops to write, so as you can see the most important effects and platforms will be targeted first as we make our way through them.
Firelight Technologies FMOD Ex
**FMOD DESIGNER NETWORK API PROGRAMMER'S TUTORIAL**

**Introduction.**

This section provides instructions on how to use the FMOD Designer Network API. By using the FMOD Designer Network API, it is possible to use the FMOD Designer tool to connect to your game as it is running and tweak the properties of events as they're playing. This is useful for sound designers as they can, for example, play the game and adjust volume levels of events in realtime as they hear them.
Using the NetEventSystem functions.

To use the FMOD Designer Network API all you need to do is:

- Call `NetEventSystem_Init` and pass it a pointer to your EventSystem object.
- Call `NetEventSystem_Update` just after each call you make to `EventSystem::update`.
- Call `NetEventSystem_Shutdown` after you call `EventSystem::release`.
- Link with fmod_event_net.lib as well as fmod_event.lib.
**Connecting to your game using FMOD Designer.**

When your game is up and running using the FMOD Designer Network API, you can use the FMOD Designer tool to connect to it at any time.

To connect to your game:

- Select "Audition -> Manage Connections..." from the menu.
- Add a new connection and fill in the relevant details. You can use the loopback address 127.0.0.1 if you want to run your game and FMOD Designer on the same machine.
- Click "Connect".
- Load a project containing some or all of the events that will play in the game. Note: FMOD Designer can't trigger new events within your game. If you want to hear the results of your tweaking, you must to trigger the relevant events from the game side.
- Now you can adjust the properties of event using the event property sheet in the event view, and envelopes / settings in event editor view.
- Save your changes in FMOD Designer using "File -> Save" at any time. Note: You must build the project again before these changes are made permanent in the .FEV file.
Firelight Technologies FMOD Ex
FMOD Ex and movie players

Introduction

This section describes how to have FMOD happily coexist with various movie playback systems available.
Windows

If you need to get a handle to FMOD's internal DirectSound pointer, you can share it by calling `System::getOutputHandle` and casting it to the appropriate pointer type. Windows allows multiple directsound interfaces to be created, so this may not be necessary.
XBox 360

If you need to get a handle to FMOD's internal XAudio2 pointer to pass to a movie player, you can share it by calling `System::getOutputHandle` and casting it to the appropriate pointer type. If the movie player has already initialized XAudio2, you can stop FMOD from initializing XAudio2 by using `FMOD_360_EXTRADRIVERDATA` in `fmodxbox360.h`, and pass the existing XAudio2 handle to 'xaudio2instance'. 
Playstation 3

A movie player can generally create its own libaudio output port and it will run along side FMOD.
If it does not work when sharing a SPURS instance, it may be because of the SPURS mode FMOD is running in.
By default, FMOD uses only one SPURS task for both the mixer and stream decoding. This SPURS task is created with no context, so cannot swap out.
If you look in fmodps3.h, you will see the FMOD_PS3_SPURSMODE enumerations. You can use these to set the SPURS behaviour in FMOD_PS3_EXTRADRIVERDATA.

There are three modes,

- **FMOD_PS3_SPURSMODE_NOCONTEXT** - described above, which is the default.
- **FMOD_PS3_SPURSMODE_CREATECONTEXT** - same as above, but created with a context, so hence can be swapped out.
- **FMOD_PS3_SPURSMODE_TWOTASKS** - Mode that used to be used in 4.26 and below by default. This uses 2 SPURS tasks, one for mpeg stream decoding and one for mixing.

If the movie player has initialized libAudio, go to fmodps3.h again and set cell_audio_initialized from the FMOD_PS3_EXTRADRIVERDATA structure to TRUE. This will stop FMOD from trying to reinitialize it.
You then define the audio port number used with the cell_audio_port member of the FMOD_PS3_EXTRADRIVERDATA structure.
If the movie player has initialized AX, go to fmodwii.h and set audiolibsinitialized from the FMOD_PS3_EXTRADRIVERDATA structure to TRUE. This will stop FMOD from trying to reinitialize it. Otherwise the movie player must not initialize AI/AX/MIX if FMOD has already done so first.
PlayStation 2.

This section describes how to have FMOD happily coexist with various movie playback systems available. The main causes on conflicts between other middleware that uses audio or the IOP, and FMOD are:

**Conflict between the SPU2 DMA channels.** There are 2 of these. DMA channel 0 and DMA channel 1. By default FMOD uses SPU2 DMA channel 0 for software mixing, and DMA core 1 for uploading sample data and for streaming to. This means DMA core 1 is used when System::createSound is being executed to load a PS2 FSB file, or streaming using System::createStream. To work around this issue see the following tips.

- **Turn off the FMOD software mixer.** This is already done if you are using the _reduced version of the library. This will free up DMA Channel 0. Most of the time you are not going to need the FMOD software mixer. You can do this by using System::init with the FMOD_INIT_DISABLESOFTWARE flag.
- **Swap FMOD's mixer/upload channel usage around.** If the 3rd party software still uses DMA Channel 1 (the channel FMOD uses for bank uploads and streaming), you can either change your 3rd party software to use DMA Channel 0 instead of DMA Channel 1, or tell FMOD to swap its usage around by specifying FMOD_INIT_PS2_SWAPDMACHANNELS. If you didn't turn the software mixer off, this would make FMOD use DMA Channel 0 for streams and sample bank uploads, and DMA Channel 1 for the software mixer.

**Conflict on the SIFCMD ports.** If your middleware or your own code is using the SIFCMD sony library to communicate with the IOP, then if you don't take care to share the SIFCMD buffers and ports with FMOD, messages will get lost and unexpected behaviour will occur in FMOD and your 3rd party software.

- **If you want to initialize your 3rd party software after FMOD.** Use this information if the code has a way to set up its SIFCMD usage. Note that FMOD uses SIFCMD port 0, 1 and 2, and has a buffer size of 16.
- **If you want to initialize your 3rd party software before FMOD.** If FMOD is initialized first, call System::init and use FMOD_PS2_EXTRADRIVERDATA structure from fmodps2.h. Also load your FMODEX.IRX or FMODEXD.IRX
with command line parameters to allow the IOP side to get the same information. For more detailed information on this see the comment above the FMOD_PS2_EXTRADRIVERDATA declaration in `fmodps2.h`.

**SPU2 ram usage and SPU2 hardware voice usage.** Because FMOD and the middleware might not know about each other, they might allocate memory or use SPU2 voices without any regard for the other.

- **Use** FMOD_SPU2_Alloc / FMOD_SPU2_GetRawAddress / FMOD_SPU2_Free. Use these functions to set aside SPU2 ram for other middleware usage.
- **Use** FMOD_SPU2_ReserveVoice. Use this function to set aside an SPU2 voice for other middleware usage.

**Bink on PS2**

Here is a quick way to get FMOD PS2 and Bink to co-exist.

Initialize FMOD first.

```
FMOD_System_Init(system, numchannels, FMOD_INIT_PS2_SWAPDMACHANNELS, ...
```

Initialize Bink

```
MovieBuffer = sceSifAllocIopHeap( RAD_IOPMemoryAmount( RAD_IOP_NEED_ RAD_IOPMemoryAmount(RAD_IOP_NEED_CORE1) ));
if ( !RAD_IOPStartUp( 1, 3, MovieBuffer, RAD_IOP_NEED_CORE1|RAD_IOP_ {
    sceSifFreeIopHeap(MovieBuffer);
    // error
}
RAD_IOPHardwareVolumes(1, 0x3fff, 0x3fff, 0x3fff, 0x3fff);
BinkSoundUseRAD_IOP(1);```
In order to co-exist with movie players, libWave needs to be used for audio output. This output mode is not as efficient as the Simple Audio output mode (default in FMOD).

What you can do is switch FMOD to use libWave on the fly when a movie needs to be played, and after the move is finished, switch FMOD back to Simple Audio. This can be done using the `FMOD_PSP_ChangeOutputMode` function declared in `fmodpsp.h`. 

- **Playstation Portable**
The main causes on conflicts between other middleware that uses audio on the XBox, and FMOD are:

"DSOUND: CMcpxAU::AllocateVoices: Error: Not enough free hardware voices". By default FMOD assumes it is in total control of the audio, so it allocates every XBox audio voice. If another 3rd party software application tries to allocate a hardware voice it will fail.

- Use `System::setHardwareChannels`. To get around this issue just call `System::setHardwareChannels` to reduce the count. XBox has around 192 HW2D voices so you could reduce this and still have plenty of voices free.
- **DSP Image incorrect.** FMOD has an internal MCP DSP image that it loads. It is stripped down to save memory, (hundreds of kilobytes) by removing unnescessary features. This may conflict with other 3rd party software that relies on a standard DSP image such as Microsoft's dsstdfx.bin.

- **Use FMOD_SpecifyEffectsImage.** This functionality is not available at this time. Contact support.
- **Make the other software use FMOD's image.** This functionality is not available at this time. Contact support.
- **Needs access to the XBox LPDIRECTSOUND handle.** If the 3rd party software is initialized second, it may want to use the XBox LPDIRECTSOUND handle.

- **Use System::getOutputHandle.** If you need to get a handle to FMOD's internal DirectSound pointer, you can share it by calling `System::getOutputHandle` and casting it to the appropriate pointer type.
Firelight Technologies FMOD Ex
THREADS AND THREAD SAFETY

Introduction

This section will talk about the threads FMOD creates, and thread safety.
FMOD threads types.

FMOD has 4 main threads, and 2 of which are created at the time of System::init, and 2 of which are created only when you use certain flags. They are

- **Mixer thread.** Software mixing thread, created at System::init.
- **Stream thread.** Thread used for decoding streams. Created the first time a sound is loaded as a stream in System::createSound with FMOD_CREATESTREAM or System::createStream.
- **Async loading thread.** Created the first time a sound is loaded with the FMOD_NONBLOCKING flag in System::createSound.
- **File reading thread.** Thread used for reading from disk for streams, to then be decoded (decompressed) by the Stream thread. Created the first time a sound is loaded as a stream in System::createSound with FMOD_CREATESTREAM or System::createStream.

Exceptions.

If FMOD_INIT_STREAM_FROM_UPDATE is used, then the stream thread will not be created.
If FMOD_OUTPUTTYPE_WAVWRITER_NRT or FMOD_OUTPUTTYPE_NOSOUND_NRT are used, then the mixer thread will not be created.

Everything else is run from the main / game thread, including System::update calculations.
- FMOD threads priorities per platform.

For reference, for your own thread code, here are the thread priorities used in FMOD per platform.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Mixer thread</th>
<th>Stream thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win32/Win64</td>
<td>THR SảnPriority_Time_Critical</td>
<td>THREAD_</td>
</tr>
<tr>
<td>(SetThreadPriority)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux/Linux64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(pthread_setschedparam)</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Mac/Mac86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(MPSetTaskWeight)</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>PS2 (EE, CreateThread)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PSP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sceKernelCreateThread)</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>PS3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sys_ppu_thread_create)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Xbox</td>
<td>THREAD_PRIORITY_TIME_CRITICAL</td>
<td>THREAD_</td>
</tr>
<tr>
<td>(SetThreadPriority)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xbox 360 (HW Thread 4, SetThreadPriority)</td>
<td>THREAD_PRIORITY_TIME_CRITICAL</td>
<td>THREAD_</td>
</tr>
<tr>
<td>Gamecube</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(OSCreateThread)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Wii</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FMOD callbacks

FMOD File and memory callbacks can possibly be called from an FMOD thread. Remember that if you specify file or memory callbacks with fmod, to make sure that they are thread safe. FMOD may call these callbacks from the stream thread, or FMOD_NONBLOCKING thread at any time.
• Calling FMOD commands from different threads.

Do not call FMOD commands from different threads! This will lead to instability, corruption and possible crashes.

Some people are tempted to put System::update into a separate thread, Only do this if you criticalsection this and all other calls to fmod.
To make FMOD thread safe would involve wrapping every FMOD function in a critical section, which adds unnecessary overhead, so at this time FMOD Ex is remaining 'not thread safe' for the time being.
Firelight Technologies FMOD Ex
REVERB NOTES

Introduction

This section will discuss FMOD's reverb parameters focusing on the software (SFX) implementation.

The fields of FMOD_REVERB_PROPERTIES (found in 'fmod.h') control both hardware and software instances of reverb. Hardware reverbs have a few parameters that the software effect doesn't use, so you can ignore these for the purpose of this discussion. For example, EnvDiffusion is NOT used by the software reverb because it is only meaningful to Wii.

FMOD's software reverb DSP is controlled by parameters defined in the I3DL2 guidelines, which describe the reverberant environment of the listener.

Here's a list of the fields of FMOD_REVERB_PROPERTIES that currently have an effect and a description of what they do within the context of the software reverb. The descriptions are much the same as in 'fmod.h'.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>Turns the reverb off if set to -1</td>
</tr>
<tr>
<td>Room</td>
<td>Room effect level at mid frequencies</td>
</tr>
<tr>
<td>RoomHF</td>
<td>Relative room effect level at high frequencies</td>
</tr>
<tr>
<td>RoomLF</td>
<td>Relative room effect level at low frequencies</td>
</tr>
<tr>
<td>DecayTime</td>
<td>Reverberation decay time at mid frequencies</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DecayHFRatio</td>
<td>High-frequency to mid-frequency decay time ratio</td>
</tr>
<tr>
<td>Reflections</td>
<td>Early reflections level relative to room effect</td>
</tr>
<tr>
<td>ReflectionsDelay</td>
<td>Initial reflection delay time</td>
</tr>
<tr>
<td>Reverb</td>
<td>Late reverberation level relative to room effect</td>
</tr>
<tr>
<td>ReverbDelay</td>
<td>Late reverberation delay time relative to initial reflection</td>
</tr>
<tr>
<td>HFReference</td>
<td>Reference high frequency (Hz) [see RoomHF]</td>
</tr>
<tr>
<td>LFReference</td>
<td>Reference low frequency (Hz) [see RoomLF]</td>
</tr>
<tr>
<td>Diffusion</td>
<td>Echo density in the late reverberation decay</td>
</tr>
<tr>
<td>Density</td>
<td>Modal density in the late reverberation decay</td>
</tr>
</tbody>
</table>
Please note:

RoomRolloffFactor is a part of I3DL2, but has no effect within FMOD.

As with most reverberation models, the response is split into sections. This implementation has early reflections and late reverberation, each of which are composed of sets of delay lines having different delay and decay characteristics.

There are a few things to note here:

1) Room, RoomHF, RoomLF, Reflections and Reverb are all measured in milliBels, i.e. 100th of a deciBel, and they're all integers.
2) Room is the input gain
3) Reflections is a gain on the output of the early reflections subsystem
4) Reverb is a gain on the output of the late reverb subsystem
5) RoomLF and LFReference control a low frequency shelving filter on the input
6) RoomHF and HFReference control a high frequency shelving filter on the input
7) Diffusion and Density control the correlation among delay lines in the reverb subsystem

The FMOD_PRESET_* presets can be useful as examples of how these parameters change the nature of the reverb. They give quite a wide scope for representing different environments.

One more thing - it's important to distinguish between FMOD_REVERB_PROPERTIES and FMOD_REVERB_CHANNELPROPERTIES. The latter is just used for controlling the a channel's input gain to the reverb, and doesn't affect the characteristics of the reverb unit itself.
Firelight Technologies FMOD Ex
COMPRESSION QUALITY, MULTICHANNEL AND LOOPING WITH LOSSY AUDIO FORMATS.

Quality and Bit rate.

What is the relationship between bit rate and the 'compression quality' property?

Within FMOD Designer, the compression quality property is found in the wave bank property panel. In FSBankEx the quality property is in the format options. The relationship between bit rate and the compression quality property (when dealing with constant bit rate compression), is appropriately:

\[ \text{bit rate} = \text{quality} \times 3.2 \]

This is the case for MP2/MP3 but may differ for XMA and other bitrate based formats.
Bit rates and sample rates for MPEG data

The following table shows the available bit rates and sample rates available for MPEG data within FMOD:

**Note!** This is the MPEG version, not the 'layer' version. Layer 2 and 3 are commonly known as MP2/MP3. MP3 for example could be MPEG 1 or 2, but is still 'layer 3'
Both MP2 support and MP3 support share the same MPEG versions and bitrate/samplerate capabilities.

<table>
<thead>
<tr>
<th>MPEG 1 Bitrates (kbps)</th>
<th>MPEG 1 Sample rates (kHz)</th>
<th>MPEG 2 Bitrates (kbps)</th>
<th>MPEG 2 Sample rates (kHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>32</td>
<td>8*</td>
<td>8*</td>
</tr>
<tr>
<td>48</td>
<td>44.1</td>
<td>16</td>
<td>11.025*</td>
</tr>
<tr>
<td>56</td>
<td>48</td>
<td>24</td>
<td>12*</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>40</td>
<td>22.05</td>
</tr>
<tr>
<td>96</td>
<td></td>
<td>48</td>
<td>24</td>
</tr>
<tr>
<td>112</td>
<td></td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td></td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>192</td>
<td></td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>224</td>
<td></td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>320</td>
<td></td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>384</td>
<td></td>
<td>160</td>
<td></td>
</tr>
</tbody>
</table>

* Note that the crossed out values are not supported by FSBankEx even though they are specified as part of the MPEG format specification.

Should the user attempt to use a sample rate not listed, FMOD will automatically
resample the file (upwards) to the next valid sample rate. For example, a file with a sample rate of 15kHz will be resampled to 16Hz.
Multi-channel MPEG Encoding

FMOD is able to create MPEG files with up to 16 channels (eight stereo pairs). To do this, the build process:

- Encodes each stereo pair into fixed sized MPEG frames. The size of the frames is determined by the bit rate. The size of the frame must be a multiple of 16 bytes. To insure this, a pad of 0 to 15 bytes is placed at the end of each frame.
- Interleaves a frame from each stereo pair into a multi-channel frame.

This process is illustrated in the figure below.

**Figure 1: Encoding a multi-channel MPEG file**

For example, let's consider a six-channel MPEG file using a constant bit rate of 128 kbps. The six channels are encoded into three stereo pairs. Each frame of stereo MPEG data is 432 bytes (including a 14 byte buffer). FMOD interleaves the stereo frames every 432 bytes into a multi-channel MPEG frame. The size of the multi-channel MPEG frame can be calculated as frame size * Number of stereo pairs. In this example, the multi-channel MPEG frame is 432 * 3, giving 864 bytes.
**Encoding mp3 files for seamless looping**

Typically when an mp3 file is looped, an audible gap can be heard when playback loops back to the start. This gap is obvious when the loop requires a sample accurate stitching from the last sample to the first. This occurs for a number of reasons, the two major factors being:

- MPEG 1 layer 3 encodes the audio data into frames of 1152 samples. If the audio data doesn't fill a frame (most importantly the last frame), the encoder will pad the frame with silent samples (some encoder will add an entire silent frame!)
- The decoding of an mp3 frame is dependent on the previous frame. When a loop occurs, the decoder will require data from the last frame to smoothly loop back to the first frame.

Without special encoding, it is not possible for mp3 data to loop seamlessly - fortunately FMOD does provide a method to do just that! The FMOD mp3 encoder can be accessed via FMOD Designer or FSBankEx. For Designer users, the special encoder is automatically used if the sound definition instance is set to loop and the wave bank compression property to 'MP3'. **Note:** if the sound definition instance is set to 'one-shot' the standard mp3 encoding is used. Users of the lower level API can specify the FSBankEx to encode mp3 data for seamless looping.

So what does FMOD do to provide seamless loop of mp3 data?

Firstly, FMOD's encoder will resample and stretch the last frame to ensure that all 1152 samples of the frame are used. This will ensure the frame is not padded with silent samples.

When used on some sources, this process may cause a slightly audible pitch change artifact. If this is the case, user are encouraged to repeat the audio within the file to increase the file size, so the time stretch distance becomes less significant. Users may also resize the length of their audio to a multiple of the frame size. The table below lists the frame size for various formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>Frame size (samples)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
With the removal of any padding within the last frame, FMOD's encoder must then prime the first frame with data from the last frame. The last frame is then removed. This allows FMOD's decoder to avoid issues of frame dependency between the first and last frame and provide a seamless loop.

In most situations FMOD's encoder and decoder will perform perfect looping of mp3 content. However some audible artifacts can be introduced, this is illustrated below.

Figure 2: Encoding MPEG frames for seamless looping

When the first frame contains silence and the last frame contains an audible signal, the interpolation used in priming the first frame will result in an audible 'pop'. Should users require silence in the first frame of their loop, they should:

- make sure the original wave loops properly, or
- pad the end of the file with a frame of silence.
• **XMA Quality and Compression**

As specified (in part) in the Xbox SDK documentation:

• The XMA encoder allows the Sound Designer to specify a quality setting between 1 and 100, where:
  
  ◦ 1 provides the highest compression level and the lowest quality, and
  ◦ 100 provides the lowest compression level and the highest quality.

• XMA’s variable bit rate compression is content dependent, meaning compression ratios can vary greatly between pieces of content.
• This means the quality settings do not translate directly to specific compression ratios.
• The Xbox 360 Development Kit suggests a compression rate between 8:1 and 15:1 will provide adequate quality for most game audio assets.
Firelight Technologies FMOD Ex
Using multiple reverbs

Introduction

In some situations, multiple styles of reverberations within a single environment must be modeled. For example, imagine a large church hall with a tunnel down into the catacombs. The reverb applied to the player's footsteps within the church hall (such as `FMOD_PRESET_STONEROOM`) could be quite different to that of the monster sounds emitting from the tunnel (which may be applied with both `FMOD_PRESET_SEWERPIPE` and `FMOD_PRESET_STONEROOM`). To handle this situation, multiple instances of the reverb DSP are required. As many as four instances of the reverb DSP can be added to the `FMOD Ex DSP Network` (at a cost of more CPU time and memory usage).

Note: Should you want to model multiple reverb types within an environment without the extra resource expense of multiple physical reverbs, see the 3D reverb tutorial, which covers automated 3D reverb zones using single reverb instance.

In this section we will look at examples of:

- Adding physical reverbs using the low level API (with some additional suggestions for the Event System)
- Querying an instance's reverb properties
- Controlling the wet/dry mix of each reverb instance per channel
Setting up the reverbs

Below is an example of setting up four reverb instances. You do not need to explicitly the extra reverb instances - the FMOD Ex engine creates them and connects them to the DSP Network when you reference them.

In the following example we will use `System::setReverbProperties` to specify four different reverb effects. If using the event system, you can simply substitute the `System::setReverbProperties` method call for `EventSystem::setReverbProperties`.

First we define four different `FMOD_REVERB_PROPERTIES` structures. The example below uses presets. You can define your own reverb settings but presets make it easier to get some common reverbs working.

```cpp
FMOD_REVERB_PROPERTIES prop1 = FMOD_PRESET_HALLWAY;
FMOD_REVERB_PROPERTIES prop2 = FMOD_PRESET_SEWERPIPE;
FMOD_REVERB_PROPERTIES prop3 = FMOD_PRESET_PARKINGLOT;
FMOD_REVERB_PROPERTIES prop4 = FMOD_PRESET_CONCERTHALL;
```

We then change the 'Instance' member to set which reverb DSP unit will be used for each preset, then call the `System::setReverbProperties` function.

```cpp
prop1.Instance = 0;
result = system->setReverbProperties(&prop1);
prop2.Instance = 1;
result = system->setReverbProperties(&prop2);
prop3.Instance = 2;
result = system->setReverbProperties(&prop3);
prop4.Instance = 3;
result = system->setReverbProperties(&prop4);
```
Getting the REVERB properties

Should you wish to get the current System reverb properties, you must first set the Instance member before calling System::getReverbProperties. In this example we will get the properties for Instance 3.

```c
FMOD_REVERB_PROPERTIES prop = { 0 };  
prop.Instance = 3;  
result = system->getReverbProperties(&prop);
```
Setting the wet/dry mix per Channel

Each channel of the FMOD Ex mixer can set their wet/dry mix for each reverb with `Channel::setReverbProperties`. In the next example we shall set both the wet and dry values of all 4 instances of reverb for a given channel. By default a channel will send to all instances. This example sets the instance 1 send value to -10000mb (off).

```
FMOD_REVERB_CHANNELPROPERTIES prop = { 0 };

Get the settings for this channel for reverb instance 1. Note that all other members are 0, only the instance flag is read when being passed to the 'get' function. Everything else will be overwritten.

prop.Flags |= FMOD_REVERB_CHANNELFLAGS_INSTANCE1;
result = channel->getReverbProperties(&prop);
```

Now change the room setting to -10,000 db to make the wet mix for this channel silent.

```
prop.Room = -10000;
result = channel->setReverbProperties(&prop);
```

Here is an alternate version, turning all send values for all 4 reverb instances to -10000mb (all off).

```
FMOD_REVERB_CHANNELPROPERTIES prop = { 0 };

Passing in a null filled structure will mean the settings for instance 0 will be returned.

result = channel->getReverbProperties(&prop);
```

Now use the same structure, and change the room value to -10,000db. Leaving other members in tact will make the wet connection silent.

```
prop.Room = -10000;
prop.Flags |= FMOD_REVERB_CHANNELFLAGS_INSTANCE0;
prop.Flags |= FMOD_REVERB_CHANNELFLAGS_INSTANCE1;
prop.Flags |= FMOD_REVERB_CHANNELFLAGS_INSTANCE2;
prop.Flags |= FMOD_REVERB_CHANNELFLAGS_INSTANCE3;
```
result = channel->setReverbProperties(&prop);

To continue the above example, the send values to reverbs 1 and 3 will be switched back on:

FMOD_REVERB_CHANNELPROPERTIES prop = { 0 };

First get the channel properties for event instance 1 so that we can use it to call with the set function. Note that if you set any instance flags, the instance settings for the first will be returned.

prop.Flags = FMOD_REVERB_CHANNELFLAGS_INSTANCE1;
result = channel->getReverbProperties(&prop);

Now change the room setting to 0db and tell the set function we want to set 2 reverb instances at once by specifying the flags for instance 1 and 3. Instance 1 flag does not need to be reset again but it is here for clarity.

prop.Room = 0;
prop.Flags |= FMOD_REVERB_CHANNELFLAGS_INSTANCE1;
prop.Flags |= FMOD_REVERB_CHANNELFLAGS_INSTANCE3;
result = channel->setReverbProperties(&prop);
Firelight Technologies FMOD Ex
Asynchronous I/O and deferred file reading

Introduction

This tutorial will describe how to defer file reading in FMOD so that you don't have to immediately satisfy FMOD's requests for data. This sort of behavior is highly desirable in game streaming engines that do not have access to the data yet, or for when accessing data out of order or in a non sequential fashion would greatly degrade performance. FMOD's asynchronous I/O callbacks will allow you to receive an FMOD read request and defer it to a later time when the game is ready. FMOD will use priorities to notify the game engine how urgent the read request is, as sometimes deferring a music stream read for example could result in stuttering audio.
Setup : Override FMOD's file system with callbacks

The idea is that you are wanting to override the file I/O that FMOD normally performs internally. You may have done this before with the
System::setFileSystem by overriding the following callbacks:

```
FMOD_FILE_OPENCALLBACK    useropen
FMOD_FILE_CLOSECALLBACK   userclose
FMOD_FILE_READCALLBACK    userread
FMOD_FILE_SEEKCALLBACK    userseek
```

The normal behavior here is that you would need to satisfy FMOD's read and seek requests immediately in a blocking fashion.
In the open callback, you open your internal file handle and return it to FMOD, along with the file size.
You would have to set all callbacks or file system override would not work. Any
callback that is null in the above callback list will cause FMOD to use the
default internal system and ignore your callbacks. All callbacks must be set.

With async I/O, there are 2 new callbacks which you can use to replace the 'userread' and 'userseek' callbacks:

```
FMOD_FILE_ASYNCREADCALLBACK  userasyncread
FMOD_FILE_ASYNCCANCELCALLBACK userasynccancel
```

If these callbacks are set, the 'userread' and 'userseek' callbacks are made redundant. You can of course keep 'userread' and 'userseek' defined if you want to switch between the 2 systems for some reason, but when 'userasyncread' is defined, the normal read/seek callbacks will never be called.
Defining the basics - opening and closing the file handle.

Before we start, we'll just define the open and close callback. A very simple implementation using stdio is provided below:

```c
FMOD_RESULT F_CALLBACK myopen(const char *name, int unicode, unsigned int* filesize, void **handle, void **userdata)
{
    if (name)
    {
        FILE *fp;
        fp = fopen(name, "rb");
        if (!fp)
        {
            return FMOD_ERR_FILE_NOTFOUND;
        }
        fseek(fp, 0, SEEK_END);
        *filesize = ftell(fp);
        fseek(fp, 0, SEEK_SET);
        *userdata = (void*)0x12345678;
        *handle = fp;
    }
    return FMOD_OK;
}

FMOD_RESULT F_CALLBACK myclose(void *handle, void *userdata)
{
    if (!handle)
    {
        return FMOD_ERR_INVALID_PARAM;
    }
    fclose((FILE*)handle);
    return FMOD_OK;
}
```
Defining 'userasyncread'

The idea for asynchronous reading, is that FMOD will request data (note, possibly from any thread - so be wary of thread safety in your code!), but you don't have to give the data to FMOD immediately. You can return from the callback without giving FMOD any data. This is deferred I/O.

For example, here is a definition of an async read callback:

```c
FMOD_RESULT F_CALLBACK myasyncread(FMOD_ASYNCREADINFO *info, void *u
{        return PutReadRequestOntoQueue(info);
}
```

Note that we didn’t actually do any read here. You can return immediately and FMOD will internally wait until the read request is satisfied. Note that if FMOD decides to wait from the main thread (which it will do often), then you cannot satisfy the queue from the main thread, you will get a deadlock. Just put the request onto a queue. We'll discuss how to let FMOD know that the data is ready in the next section.

There are a few things to consider here:

- The callback could come from any thread inside FMOD's system. Usually this means FMOD's streaming thread, FMOD's file I/O thread, the main thread, or the FMOD_NONBLOCKING thread. Be thread safe! Use critical sections around linked list/queue operations to avoid corruption of data.
- Return code. This is usually a fatal, non disk related error such as not being able to add to the queue. This could be an out of memory error for example. Use FMOD_ERR_MEMORY as the return value if this is the case. Return FMOD_OK in normal cases. It normally won't be a return code related to a disk error. You have to set the 'result' code in the FMOD_ASYNCREADINFO structure to let FMOD know about a file based error.
- Be wary that your queued command may need to be cancelled if the user decides to release the FMOD resource that is using that file, such as a sound. See the next section about myasynccancel in that case.
- The FMOD_ASYNCREADINFO structure is where you fill in the data requested by FMOD. See below for a more detailed description of this structure and what is required to complete the read.
Defining 'userasynccancel'

If you have queued up a lot of read requests, and have not satisfied them yet, then it is possible that the user may want to release a sound before the request has been fulfilled (ie Sound::release is called). In that case FMOD will call the async cancel callback to let you cancel any operations you may have pending, that are related to this file.

```c
FMOD_RESULT F_CALLBACK myasynccancel(void *handle, void *userdata)
{
    return SearchQueueForFileHandleAndRemove(info);
}
```

Note that the above callback implementation will search through our internal linked list (in a thread safe fashion), removing any requests from the queue so that they don't get processed after the Sound is released. If it is in the middle of reading, then the callback will wait until the read is finished and then return. Do not return while a read is happening, or before a read happens, as the memory for the read destination will be freed and the deferred read will read into an invalid pointer.
• **Filling out the FMODASYNCREADINFO structure when performing a deferred read**

The **FMODASYNCREADINFO** is the structure you will pass to your deferred I/O system, and will be the structure that you read and fill out when fulfilling the requests.

The structure exposes the features of the async read system. These are:

• Priority is supported. FMOD will let the user know if the read is not important, mildly important, or extremely important. This will allow the user to reschedule the queue to make important reads happen before non-important reads.
• Read completion is signalled by simply setting the 'result' code of **FMODASYNCREADINFO**.
• Memory does not need to be copied anywhere, you can read directly into FMOD's pointers which point directly to the internal file buffers.
• You do not have to give FMOD all of the data, you can give a partial read result to the callback and FMOD will most likely just issue another read request later with a smaller byte value.

```c
typedef struct {
    void * handle;
    unsigned int offset;
    unsigned int sizebytes;
    int priority;
    void * buffer;
    unsigned int bytesRead;
    FMOD_RESULT result;
    void * userdata;
} FMODASYNCREADINFO;
```

The first 4 members (**handle, offset, sizebytes, priority**) are read only values, which tell you about the file handle in question, where in the file it wants to read from (so no seek callbacks required!) and how many bytes it wants. The priority value tells you how important the read is as discussed previously.

The next 3 members (**buffer, bytesRead and result**) are values you will fill in, and to let FMOD know that you have read the data.
Read your file data into buffer. sizebytes is how much you should be reading. bytesread is how much you actually read (this could be less than sizebytes). If you hit the 'end of file' condition and need to return less bytes than were requested - set bytesread to less than sizebytes, and then set the result to FMOD_ERR_FILE_EOF.

**Set the result last!**

Note! Do not set the result before setting the bytesread value and reading the data into buffer. The initial value for result, is going to be FMOD_ERR_NOTREADY. When you set the value to FMOD_OK (or appropriate error code) then internally FMOD will immediately see this as an indication to continue, so if the bytesread or buffer contents are not ready, you will get corruption, errors or unexpected behaviour.

So to summarize, the last thing you will do before finishing your queue process is to set result. You will not set it before setting bytesread or filling in buffer.
• **Threading issues & read priorities**

As mentioned earlier in this tutorial, FMOD can call the read callback from various different threads, so it is common sense to protect your I/O system from operations happening simultaneously from different threads.

A system that would use FMOD's async I/O feature would most likely be running in its own thread. This is so the blocking wait loops in FMOD's loading calls are not forever waiting for data because the user can't provide it to FMOD. If the system runs in another thread, it can detect the queue insert, and process the data while FMOD is waiting.

It is actually possible to complete the read as if it wasn't deferred, and do a direct file read into the buffer and set sizebytes/result values from the FMOD async read callback. This is a possible way to reduce delays for extremely urgent FMOD reads. Currently there are 3 different categories of read priority.

- 0 = low priority. These reads are usually blocking style reads that come from a user load command, and there are no real negative side effects of delaying the read except that the load function takes longer. These reads are going to be issued from a `System::createSound` call, or a `EventSystem::load` call for example.
- 50 = medium priority. These reads are important, and usually come from the FMOD stream system. They can be delayed, but not for too long. If the delay is too long, then audio streams will starve, and possibly stutter. If you need to delay the read longer, the FMOD file buffer size can be increased with `System::setStreamBufferSize`
- 100 = high priority. Currently the highest priority read issued by FMOD is when an audio stream loops. It must internally flush the stream buffer after a seek to loop start, and do so before the stream 'decode buffer' (the PCM double-buffer that the stream decoder decodes into) loops around and starts stuttering (this is a different buffer to the previously mentioned stream buffer. That one contains compressed file data. The decode buffer contains decompressed PCM data). The decode buffer is usually small so it is important to get the read done fast, but the user can also increase these buffers with `FMOD_CREATESOUNDEXINFO::decodebuffersize`. `FMOD_ADVANCEDEDSETTINGS::defaultDecodeBufferSize` can also be used to
set all future decode buffer sizes for all streams without having to set it every time, and is going to be used for the Event System because decode buffer size is not something you can set for events individually.
Firelight Technologies FMOD Ex
SRS Surround sound

How to enable SRS 5.1 Surround Sound encoding in FMOD

To enable 5.1 surround through a stereo analog output for decoding with Prologic and SRS compatible hardware, use `System::setSpeakerMode` with `FMOD_SPEAKERMODE_SRS5_1_MATRIX` as the parameter, before init.
License for SRS 5.1 Surround Sound Encoder to Customers

1. About SRS 5.1 Surround Sound Encoder

The SRS 5.1 Surround Sound Encoder for FMOD features SRS’ advanced matrix surround sound processing engine. The SRS surround engine is a highly versatile multichannel audio encode system capable of supporting a wide range of surround sound creation and playback applications. With SRS 5.1 Surround Sound, surround sound is made available without sacrificing the 2-channel reach and experience.

2. Overview of SRS 5.1 Surround Sound

SRS 5.1 Surround Sound Encoder provides the capability to encode up to 6.1 channels of audio for transmission or storage over two output channels or standard two-channel carriers. SRS 5.1 Surround Sound Encoder can be listened to in stereo or decoded TVs, PCs, receivers, mobile phones and other CE devices with SRS technologies. SRS 5.1 Surround Sound is also backward compatible with all matrix decoders, such as Dolby Pro Logic® and Dolby Pro Logic Il®, with playback performance subject to the limitations of the specific decoder.

For more information on producing content in SRS 5.1 Surround Sound, please visit the SRS website at: http://surround.srslabs.com/

For technical questions on producing content in SRS 5.1 Surround Sound, please e-mail to SRS at contentsupport@srslabs.com

3. Rights to Use of the SRS 5.1 Surround Sound Encoder and Logo

Please note that, to use the SRS 5.1 Surround Sound Encoder included by FMOD, you are required to register with SRS at http://surround.srslabs.com/. To register with SRS, you will be required to provide your company name, contact information and title(s) of games developed using the SRS 5.1 Surround Sound
Encoder if available. Your information will enable SRS to better support your technical requirements as needed and if applicable, refer to your game title(s) in SRS’ marketing materials.

In the same registration web page, you will find a trademark license agreement for your right to use the SRS 5.1 Surround Sound Logo. You must accept the trademark license agreement prior to using the SRS 5.1 Surround Sound Logo.
Firelight Technologies FMOD Ex
For Windows-specific information, please see "Getting started with FMOD for Windows.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
LINUX SPECIFIC ISSUES / FEATURES

For Linux-specific information, please see "Getting started with FMOD for Linux.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
MAC SPECIFIC ISSUES / FEATURES

For Mac-specific information, please see "Getting started with FMOD for Mac.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
PLAYSTATION 2 SPECIFIC
ISSUES / FEATURES

For PlayStation 2-specific information, please see "Getting started with FMOD for PlayStation 2.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
PLAYSTATION 3 SPECIFIC ISSUES / FEATURES

For PlayStation 3-specific information, please see "Getting started with FMOD for PlayStation 3.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
PLAYSTATION PORTABLE
SPECIFIC ISSUES / FEATURES

For PlayStation Portable-specific information, please see "Getting started with FMOD for PlayStation Portable.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
XBOX 360 SPECIFIC ISSUES / FEATURES

For Xbox 360-specific information, please see "Getting started with FMOD for Xbox 360.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
Wii SPECIFIC ISSUES / FEATURES

For Wii-specific information, please see "Getting started with FMOD for Wii.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
IPHONE SPECIFIC ISSUES / FEATURES

For iPhone/iPad-specific information, please see "Getting started with FMOD for iPhone.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
3DS SPECIFIC ISSUES / FEATURES

For 3DS specific information, please see "Getting started with FMOD for 3DS.pdf" in the documentation directory.
Firelight Technologies FMOD Ex
API Reference

FMOD Ex API Reference  FMOD Designer API Reference  
FMOD Designer Network API Reference
Firelight Technologies FMOD Ex
C++ Reference

Interfaces  Functions
Callbacks  Structures
Defines  Enumerations
Firelight Technologies FMOD Ex
Interfaces

System  Sound
Channel
ChannelGroup
SoundGroup
DSP
DSPConnection
Geometry
Reverb
Firelight Technologies FMOD Ex
System Interface

System::addDSP  System::attachFileSystem
System::close
System::createChannelGroup
System::createDSP
System::createDSPByPlugin
System::createDSPByType
System::createGeometry
System::createReverb
System::createSound
System::createSoundGroup
System::createStream
System::get3DListenerAttributes
System::get3DNumListeners
System::get3DSettings
System::get3DSpeakerPosition
System::getAdvancedSettings
System::getCDROMDriveName
System::getCPUUsage
System::getChannel
System::getChannelsPlaying
System::getDSPBufferSize
System::getDSPClock
System::getDSPHead
System::getDriver
System::getDriverCaps
System::getDriverInfo
System::getDriverInfoW
System::getGeometryOcclusion
System::getGeometrySettings
System::getHardwareChannels
System::getMasterChannelGroup
System::getMasterSoundGroup
System::getMemoryInfo
System::getNetworkProxy
System::getNetworkTimeout
System::getNumCDROMDrives
System::getNumDrivers
System::getNumPlugins
System::getOutput
System::getOutputByPlugin
System::getOutputHandle
System::getPluginHandle
System::getPluginInfo
System::getRecordDriverCaps
System::getRecordDriverInfo
System::getRecordDriverInfoW
System::getRecordNumDrivers
System::getRecordPosition
System::getReverbAmbientProperties
System::getReverbProperties
System::getSoftwareChannels
System::getSoftwareFormat
System::getSoundRAM
System::getSpeakerMode
System::getSpectrum
System::getStreamBufferSize
System::getUserData
System::getVersion
System::getWaveData
System::init
System::isRecording
System::loadGeometry
System::loadPlugin
System::lockDSP
System::playDSP
System::playSound
System::recordStart
System::recordStop
System::registerCodec
System::registerDSP
System::release
System::set3DListenerAttributes
System::set3DNumListeners
System::set3DRolloffCallback
System::set3DSettings
System::set3DSpeakerPosition
System::setAdvancedSettings
System::setCallback
System::setDSPBufferSize
System::setDriver
System::setFileSystem
System::setGeometrySettings
System::setHardwareChannels
System::setNetworkProxy
System::setNetworkTimeout
System::setOutput
System::setOutputByPlugin
System::setPluginPath
System::setReverbAmbientProperties
System::setReverbProperties
System::setSoftwareChannels
System::setSoftwareFormat
System::setSpeakerMode
System::setStreamBufferSize
System::setUserData
System::unloadPlugin
System::unlockDSP
System::update
Firelight Technologies FMOD Ex
System::addDSP

This function adds a pre-created DSP unit or effect to the head of the System DSP chain.

C++ Syntax

```cpp
FMOD_RESULT System::addDSP(
    FMOD::DSP * dsp,
    FMOD::DSPConnection ** connection
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_AddDSP(
    FMOD_SYSTEM * system,
    FMOD_DSP * dsp,
    FMOD_DSPCONNECTION ** connection
);
```

Parameters

`dsp`

A pointer to a pre-created DSP unit to be inserted at the head of the System DSP chain.

`connection`

A pointer to the connection involved between the System DSP head and the specified dsp unit. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

This function is a wrapper function to insert a DSP unit at the top of the System DSP chain. It disconnects the head unit from its input, then inserts the unit at the head and reconnects the previously disconnected input back as as an input to the new unit.

Note: The connection pointer retrieved here will become invalid if you disconnect the 2 dsp units that use it.

See Also

- System::getDSPHead
- System::createDSP
- System::createDSPByType
- System::createDSPByPlugin
- Channel::addDSP
- ChannelGroup::addDSP
- DSP::remove
Firelight Technologies FMOD Ex
System::attachFileSystem

Function to allow a user to 'piggyback' on FMOD's file reading routines. This allows users to capture data as FMOD reads it, which may be useful for ripping the raw data that FMOD reads for hard to support sources (for example internet streams or cdda streams).

C++ Syntax

```cpp
FMOD_RESULT System::attachFileSystem(
    FMOD_FILE_OPENCALLBACK useropen,
    FMOD_FILE_CLOSECALLBACK userclose,
    FMOD_FILE_READCALLBACK userread,
    FMOD_FILE_SEEKCALLBACK userseek
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_AttachFileSystem(
    FMOD_SYSTEM * system,
    FMOD_FILE_OPENCALLBACK useropen,
    FMOD_FILE_CLOSECALLBACK userclose,
    FMOD_FILE_READCALLBACK userread,
    FMOD_FILE_SEEKCALLBACK userseek
);
```

Parameters

`useropen`

Pointer to an open callback which is called after a file is opened by FMOD.

`userclose`

Pointer to a close callback which is called after a file is closed by FMOD.

`userread`

Pointer to a read callback which is called after a file is read by FMOD.

`userseek`
Pointer to a seek callback which is called after a file is seeked into by FMOD.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

NOTE! Do not use this to 'override' FMOD's file system! That is what `setFileSystem` is for. This function is purely for 'snooping' and letting FMOD do its own file access, but if you want to capture what FMOD is reading you can do it with this function.

**See Also**

- [System::setFileSystem](#)
- [FMOD_FILE_OPENCALLBACK](#)
- [FMOD_FILE_CLOSECALLBACK](#)
- [FMOD_FILE_READCALLBACK](#)
- [FMOD_FILE_SEEKCALLBACK](#)
Firelight Technologies FMOD Ex
System::close

Closes the system object without freeing the object's memory, so the system handle will still be valid. Closing the output renders objects created with this system object invalid. Make sure any sounds, channelgroups, geometry and dsp objects are released before closing the system object.

C++ Syntax

```cpp
FMOD_RESULT System::close();
```

C Syntax

```c
FMOD_RESULT FMOD_System_Close(FMOD_SYSTEM * system);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::init
- System::release
Firelight Technologies FMOD Ex
System::createChannelGroup

Creates a channel group object. These objects can be used to assign channels to
for group channel settings, such as volume.
Channel groups are also used for sub-mixing. Any channels that are assigned to
a channel group get submixed into that channel group's DSP.

C++ Syntax

```cpp
FMOD_RESULT System::createChannelGroup(
    const char * name,
    FMOD::ChannelGroup ** channelgroup
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_CreateChannelGroup(
    FMOD_SYSTEM * system,
    const char * name,
    FMOD_CHANNELGROUP ** channelgroup
);
```

Parameters

`name`
Label to give to the channel group for identification purposes. Optional (can be null).

`channelgroup`
Address of a variable to receive a newly created FMOD::ChannelGroup object.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the
FMOD_RESULT enumeration.
Remarks

See the channel group class definition for the types of operations that can be performed on 'groups' of channels.
The channel group can for example be used to have 2 separate groups of master volume, instead of one global master volume.
A channel group can be used for sub-mixing, ie so that a set of channels can be mixed into a channel group, then can have effects applied to it without affecting other channels.

See Also

- System::getMasterChannelGroup
- Channel::setChannelGroup
- ChannelGroup::release
Firelight Technologies FMOD Ex
System::createDSP

Creates a user defined DSP unit object to be inserted into a DSP network, for the purposes of sound filtering or sound generation.

C++ Syntax

```cpp
FMOD_RESULT System::createDSP(
    FMOD_DSP_DESCRIPTION * description,
    FMOD::DSP ** dsp
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_CreateDSP(
    FMOD_SYSTEM * system,
    FMOD_DSP_DESCRIPTION * description,
    FMOD_DSP ** dsp
);
```

Parameters

`description`
Address of an `FMOD_DSP_DESCRIPTION` structure containing information about the unit to be created.

`dsp`
Address of a variable to receive a newly created FMOD::DSP object.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

A DSP unit can generate or filter incoming data. The data is created or filtered through use of the read callback that is defined by the user. See the definition for the FMOD_DSP_DESCRIPTION structure to find out what each member means.

To be active, a unit must be inserted into the FMOD DSP network to be heard. Use functions such as System::addDSP, Channel::addDSP or DSP::addInput to do this.

For more information and a detailed description (with diagrams) see the tutorial on the DSP system in the documentation.

See Also

- FMOD_DSP_DESCRIPTION
- System::createDSPByType
- System::createDSPByPlugin
- System::addDSP
- Channel::addDSP
- DSP::addInput
- DSP::setActive
Firelight Technologies FMOD Ex
System::createDSPByPlugin

Creates a DSP unit object which is either built in or loaded as a plugin, to be inserted into a DSP network, for the purposes of sound filtering or sound generation.

This function creates a DSP unit that can be enumerated by using System::getNumPlugins and System::getPluginInfo.

C++ Syntax

```cpp
FMOD_RESULT System::createDSPByPlugin(
    unsigned int handle,
    FMOD::DSP ** dsp
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_CreateDSPByPlugin(
    FMOD_SYSTEM * system,
    unsigned int handle,
    FMOD_DSP ** dsp
);
```

Parameters

`handle`

Handle to a pre-existing DSP plugin.

`dsp`

Address of a variable to receive a newly created FMOD::DSP object.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Remarks

A DSP unit can generate or filter incoming data. To be active, a unit must be inserted into the FMOD DSP network to be heard. Use functions such as System::addDSP, Channel::addDSP or DSP::addInput to do this. For more information and a detailed description (with diagrams) see the tutorial on the DSP system in the documentation.

See Also

- System::getNumPlugins
- System::getPluginInfo
- System::createDSPByType
- System::createDSP
- System::addDSP
- Channel::addDSP
- DSP::addInput
- DSP::setActive
Firelight Technologies FMOD Ex
System::createDSPByType

Creates an FMOD defined built in DSP unit object to be inserted into a DSP network, for the purposes of sound filtering or sound generation. This function is used to create special effects that come built into FMOD.

C++ Syntax

```cpp
FMOD_RESULT System::createDSPByType(
    FMOD_DSP_TYPE type,
    FMOD::DSP ** dsp
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_CreateDSPByType(
    FMOD_SYSTEM * system,
    FMOD_DSP_TYPE type,
    FMOD_DSP ** dsp
);
```

Parameters

`type`

A pre-defined DSP effect or sound generator described by a `FMOD_DSP_TYPE`.

`dsp`

Address of a variable to receive a newly created FMOD::DSP object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

A DSP unit can generate or filter incoming data. To be active, a unit must be inserted into the FMOD DSP network to be heard. Use functions such as `System::addDSP`, `Channel::addDSP`, `ChannelGroup::addDSP` or `DSP::addInput` to do this. For more information and a detailed description (with diagrams) see the tutorial on the DSP system in the documentation.

**Note!** Winamp DSP and VST plugins will only return the first plugin of this type that was loaded! To access all VST or Winamp DSP plugins the `System::createDSPByPlugin` function! Use the index returned by `System::loadPlugin` if you don't want to enumerate them all.

See Also

- `FMOD_DSP_TYPE`
- `System::createDSP`
- `System::createDSPByPlugin`
- `System::addDSP`
- `System::loadPlugin`
- `Channel::addDSP`
- `ChannelGroup::addDSP`
- `DSP::addInput`
- `DSP::setActive`
Firelight Technologies FMOD Ex
System::createGeometry

Geometry creation function. This function will create a base geometry object which can then have polygons added to it.

C++ Syntax

```cpp
FMOD_RESULT System::createGeometry(
    int maxpolygons,
    int maxvertices,
    FMOD::Geometry ** geometry
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_CreateGeometry(
    FMOD_SYSTEM * system,
    int maxpolygons,
    int maxvertices,
    FMOD_GEOMETRY ** geometry
);
```

Parameters

`maxpolygons`

Maximum number of polygons within this object.

`maxvertices`

Maximum number of vertices within this object.

`geometry`

Address of a variable to receive a newly created FMOD::Geometry object.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

Polygons can be added to a geometry object using **Geometry::AddPolygon**.

A geometry object stores its list of polygons in a structure optimized for quick line intersection testing and efficient insertion and updating. The structure works best with regularly shaped polygons with minimal overlap. Many overlapping polygons, or clusters of long thin polygons may not be handled efficiently. Axis aligned polygons are handled most efficiently.

The same type of structure is used to optimize line intersection testing with multiple geometry objects.

It is important to set the value of maxworldsize to an appropriate value using **System::setGeometrySettings**. Objects or polygons outside the range of maxworldsize will not be handled efficiently. Conversely, if maxworldsize is excessively large, the structure may lose precision and efficiency may drop.

**See Also**

- **System::setGeometrySettings**
- **System::loadGeometry**
- **Geometry::AddPolygon**
Firelight Technologies FMOD Ex
System::createReverb

Creates a 'virtual reverb' object. This object reacts to 3d location and morphs the reverb environment based on how close it is to the reverb object's center. Multiple reverb objects can be created to achieve a multi-reverb environment.

**C++ Syntax**

```cpp
FMOD_RESULT System::createReverb(
    FMOD::Reverb ** reverb
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_CreateReverb(
    FMOD_SYSTEM * system,
    FMOD_REVERB ** reverb
);
```

**Parameters**

`reverb`

Address of a pointer to a Reverb object to receive the newly created virtual reverb object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

The 3D reverb object is a sphere having 3D attributes (position, minimum
distance, maximum distance) and reverb properties. The properties and 3D attributes of all reverb objects collectively determine, along with the listener's position, the settings of and input gains into a single 3D reverb DSP.

Please note that this only applies to software channels. When the listener is within the sphere of effect of one or more 3d reverbs, the listener's 3D reverb properties are a weighted combination of such 3d reverbs. When the listener is outside all of the reverbs, the 3D reverb setting is set to the default ambient reverb setting.

Use **System::setReverbAmbientProperties** to set a 'background' default reverb environment. This is a reverb that will be morphed to if the listener is not within any virtual reverb zones.

By default the ambient reverb is set to 'off'. Creating multiple reverb objects does not impact performance. These are 'virtual reverbs'. There will still be only 1 physical reverb DSP running that just morphs between the different virtual reverbs. **System::setReverbProperties** can still be used in conjunction with the 3d based virtual reverb system. This allows 2d sounds to have reverb. If this call is used at the same time virtual reverb objects are active, 2 physical reverb dps will be used, incurring a small memory and cpu hit.

Note about physical SFX Reverb DSP unit allocation. To remove the DSP unit and the associated CPU cost, make sure all 3d 3d reverb objects are released, and the ambient reverb background property is set to OFF (environment = -1). If either a 3d reverb zone is still present or the ambient setting is not off, the SFX Reverb DSP unit will remain active.

Simply creating 1 reverb object, or setting the ambient state to something other than OFF, an SFX Reverb DSP unit will be created an inserted back into the DSP network.

The user may wish to avoid a sudden cutout of reverb by leaving the ambient setting active for a while. Make sure to release all objects and set the ambient reverb setting to off to get a proper cleanup.

**See Also**

- **Reverb::release**
- **System::setReverbAmbientProperties**
• System::getReverbAmbientProperties
• System::setReverbProperties
• System::getReverbProperties
Firelight Technologies FMOD Ex
System::createSound

Loads a sound into memory, or opens it for streaming.

C++ Syntax

```cpp
FMOD_RESULT System::createSound(
    const char * name_or_data,
    FMOD_MODE mode,
    FMOD_CREATE SoundEXINFO * exinfo,
    FMOD:: Sound ** sound
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_CreateSound(
    FMOD_SYSTEM * system,
    const char * name_or_data,
    FMOD_MODE mode,
    FMOD_CREATE SoundEXINFO * exinfo,
    FMOD_Sound ** sound
);
```

Parameters

**name_or_data**

Name of the file or URL to open, or a pointer to a preloaded sound memory block if FMOD_OPENMEMORY/FMOD_OPENMEMORY_POINT is used. For CD playback the name should be a drive letter with a colon, example "D:" (windows only).

**mode**

Behaviour modifier for opening the sound. See FMOD_MODE. Also see remarks for more.

**exinfo**

Pointer to a FMOD_CREATE SoundEXINFO which lets the user provide
extended information while playing the sound. Optional. Specify 0 or NULL to ignore.

sound

Address of a variable to receive a newly created FMOD::Sound object.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Important! By default (FMOD_CREATESAMPLE) FMOD will try to load and decompress the whole sound into memory! Use FMOD_CREATESTREAM to open it as a stream and have it play back in realtime!
FMOD_CREATECOMPRESSESAMPLE can also be used for certain formats.

- To open a file or URL as a stream, so that it decompresses / reads at runtime, instead of loading / decompressing into memory all at the time of this call, use the FMOD_CREATESTREAM flag. This is like a 'stream' in FMOD 3.
- To open a file or URL as a compressed sound effect that is not streamed and is not decompressed into memory at load time, use FMOD_CREATECOMPRESSESAMPLE. This is supported with MPEG (mp2/mp3), ADPCM (wav on all platforms and vag on PSP) and XMA files only. This is useful for those who want realtime compressed sound effects, but not the overhead of disk access.
- To open a CD drive, use the drive as the name, for example on the windows platform, use "D:"
- To open a sound as 2D, so that it is not affected by 3D processing, use the FMOD_2D flag. 3D sound commands will be ignored on these types of sounds.
- To open a sound as 3D, so that it is treated as a 3D sound, use the FMOD_3D flag. Calls to Channel::setPan will be ignored on these types of sounds.
To use FMOD software mixing buffers, use the **FMODSOFTWARE** flag. This gives certain benefits, such as DSP processing, spectrum analysis, loop points, 5.1 mix levels, 2d/3d morphing, and more.

To use the soundcard's hardware to play the sound, use the **FMOD_HARDWARE** flag.

Note that **FMOD_OPENRAW**, **FMOD_OPENMEMORY**, **FMOD_OPENMEMORY_POINT** and **FMOD_OPENUSER** will not work here without the exinfo structure present, as more information is needed.

Use **FMOD_NONBLOCKING** to have the sound open or load in the background. You can use Sound::getOpenState to determine if it has finished loading / opening or not. While it is loading (not ready), sound functions are not accessible for that sound.

To account for slow devices or computers that might cause buffer underrun (skipping/stuttering/repeating blocks of audio), use System::setStreamBufferSize.

To play WMA files on Windows, the user must have the latest Windows media player codecs installed (Windows Media Player 9). The user can download this as an installer (wmfdist.exe) from www.fmod.org download page if they desire or you may wish to redistribute it with your application (this is allowed). This installer does NOT install windows media player, just the necessary WMA codecs needed.

**PlayStation 2 Note:** You can pre-pend "host0:" or "cdrom0:" if you like. FMOD will automatically add "host0:" to the filename if it is not found.

Specifying **FMOD_OPENMEMORY_POINT** will POINT to your memory rather allocating its own sound buffers and duplicating it internally

**This means you cannot free the memory while FMOD is using it, until after Sound::release is called.** With **FMOD_OPENMEMORY_POINT**, for PCM formats, only WAV, FSB and RAW are supported. For compressed formats, only those formats supported by **FMOD_CREATECOMPRESSEDSAMPLE** are supported.

**See Also**

- **FMOD_MODE**
- **FMOD_CREATENODINDEXINFO**
- **Sound::getOpenState**
- **System::setStreamBufferSize**
- **Channel::setPan**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
System::createSoundGroup

Creates a sound group, which can store handles to multiple Sound pointers.

**C++ Syntax**

```cpp
FMOD_RESULT System::createSoundGroup(
    const char * name,
    FMOD::SoundGroup ** soundgroup
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_CreateSoundGroup(
    FMOD_SYSTEM * system,
    const char * name,
    FMOD_SOUNDGROUP ** soundgroup
);
```

**Parameters**

- **name**
  
  Name of sound group.

- **soundgroup**
  
  Address of a variable to recieve a pointer to a sound group.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**
Once a SoundGroup is created, `Sound::setSoundGroup` is used to put a sound in a SoundGroup.

See Also

- `SoundGroup::release`
- `Sound::setSoundGroup`
Firelight Technologies FMOD Ex
System::createStream

Opens a sound for streaming. This function is a helper function that is the same as System::createSound but has the FMOD_CREATESTREAM flag added internally.

C++ Syntax

```cpp
FMOD_RESULT System::createStream(
    const char * name_or_data,
    FMOD_MODE mode,
    FMOD_CREATESTREAMINFO * exinfo,
    FMOD::Sound ** sound
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_CreateStream(
    FMOD_SYSTEM * system,
    const char * name_or_data,
    FMOD_MODE mode,
    FMOD_CREATESTREAMINFO * exinfo,
    FMOD_SOUND ** sound
);
```

Parameters

`name_or_data`

Name of the file or URL to open. For CD playback this may be a drive letter with a colon, example "D:".

`mode`

Behaviour modifier for opening the sound. See FMOD_MODE. Also see remarks for more.

`exinfo`

Pointer to a FMOD_CREATESTREAMEXINFO which lets the user provide
extended information while playing the sound. Optional. Specify 0 or NULL to ignore.

```
sound
```

Address of a variable to receive a newly created FMOD::Sound object.

### Return Values

If the function succeeds then the return value is **FMOD_OK**.

If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

### Remarks

Note that a stream only has 1 decode buffer and file handle, and therefore can only be played once. It cannot play multiple times at once because it cannot share a stream buffer if the stream is playing at different positions. Open multiple streams to have them play concurrently.

- To open a file or URL as a stream, so that it decompresses / reads at runtime, instead of loading / decompressing into memory all at the time of this call, use the **FMOD_CREATESTREAM** flag. This is like a 'stream' in FMOD 3.
- To open a file or URL as a compressed sound effect that is not streamed and is not decompressed into memory at load time, use **FMOD_CREATECOMPRESSEDSAMPLE**. This is supported with MPEG (mp2/mp3), ADPCM (wav on all platforms and VAG on PSP) and XMA files only. This is useful for those who want realtime compressed sound effects, but not the overhead of disk access.
- To open a CD drive, use the drive as the name, for example on the windows platform, use "D:"
- To open a sound as 2D, so that it is not affected by 3D processing, use the **FMOD_2D** flag. 3D sound commands will be ignored on these types of sounds.
- To open a sound as 3D, so that it is treated as a 3D sound, use the **FMOD_3D** flag. Calls to `Channel::setPan` will be ignored on these types of sounds.
- To use FMOD software mixing buffers, use the `FMOD_SOFTWARE` flag. This gives certain benefits, such as DSP processing, spectrum analysis, loop points, 5.1 mix levels, 2d/3d morphing, and more.
- To use the soundcard's hardware to play the sound, use the `FMOD_HARDWARE` flag.

Note that `FMOD_OPENRAW`, `FMOD_OPENMEMORY`, `FMOD_OPENMEMORY_POINT` and `FMOD_OPENUSER` will not work here without the exinfo structure present, as more information is needed.

Use `FMOD_NONBLOCKING` to have the sound open or load in the background. You can use `Sound::getOpenState` to determine if it has finished loading / opening or not. While it is loading (not ready), sound functions are not accessible for that sound.

To account for slow devices or computers that might cause buffer underrun (skipping/stuttering/repeating blocks of audio), use `System::setStreamBufferSize`.

Note that `FMOD_CREATESAMPLE` will be ignored, overridden by this function because this is simply a wrapper to `System::createSound` that provides the `FMOD_CREATESTREAM` flag. The `FMOD_CREATESTREAM` flag overrides `FMOD_CREATESAMPLE`.

See Also

- `FMOD_MODE`
- `FMOD_CREATESOUNDEXINFO`
- `Sound::getOpenState`
- `System::setStreamBufferSize`
- `System::createSound`
- `Channel::setPan`
Firelight Technologies FMOD Ex
System::get3DListenerAttributes

This retrieves the position, velocity and orientation of the specified 3D sound listener.

C++ Syntax

```cpp
FMOD_RESULT System::get3DListenerAttributes(
    int listener,
    FMOD_VECTOR * pos,
    FMOD_VECTOR * vel,
    FMOD_VECTOR * forward,
    FMOD_VECTOR * up
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_Get3DListenerAttributes(
    FMOD_SYSTEM * system,
    int listener,
    FMOD_VECTOR * pos,
    FMOD_VECTOR * vel,
    FMOD_VECTOR * forward,
    FMOD_VECTOR * up
);
```

Parameters

**listener**

Listener ID in a multi-listener environment. Specify 0 if there is only 1 listener.

**pos**

Address of a variable that receives the position of the listener in world space, measured in distance units. Optional. Specify 0 or NULL to ignore.

**vel**

Address of a variable that receives the velocity of the listener measured in distance units **per second**. Optional. Specify 0 or NULL to ignore.
forward

Address of a variable that receives the forwards orientation of the listener. Optional. Specify 0 or NULL to ignore.

up

Address of a variable that receives the upwards orientation of the listener. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `System::set3DListenerAttributes`
- `FMOD_VECTOR`
Firelight Technologies FMOD Ex
**System::get3DNumListeners**

Retrieves the number of 3D listeners.

**C++ Syntax**

```cpp
FMOD_RESULT System::get3DNumListeners(
    int * numlisteners
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_Get3DNumListeners(
    FMOD_SYSTEM * system,
    int * numlisteners
);
```

**Parameters**

`numlisteners`

Address of a variable that receives the current number of 3D listeners in the 3D scene.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [System::set3DNumListeners](#)
Firelight Technologies FMOD Ex
System::get3DSettings

Retrieves the global doppler scale, distance factor and rolloff scale for all 3D sound in FMOD.

C++ Syntax

```
FMOD_RESULT System::get3DSettings(
    float * dopplerscale,
    float * distancefactor,
    float * rolloffscale
);
```

C Syntax

```
FMOD_RESULT FMOD_System_Get3DSettings(
    FMOD_SYSTEM * system,
    float * dopplerscale,
    float * distancefactor,
    float * rolloffscale
);
```

Parameters

*dopplerscale*

Address of a variable that receives the scaling factor for doppler shift. Optional. Specify 0 or NULL to ignore.

*distancefactor*

Address of a variable that receives the relative distance factor to FMOD's units. Optional. Specify 0 or NULL to ignore.

*rolloffscale*

Address of a variable that receives the scaling factor for 3D sound rolloff or attenuation. Optional. Specify 0 or NULL to ignore.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [System::set3DSettings](#)
Firelight Technologies FMOD Ex
System::get3DSpeakerPosition

Retrieves the current speaker position information for the selected speaker.

C++ Syntax

```cpp
FMOD_RESULT System::get3DSpeakerPosition(
    FMOD_SPEAKER   speaker,
    float *   x,
    float *   y,
    bool *   active
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_Get3DSpeakerPosition(
    FMOD_SYSTEM *  system,
    FMOD_SPEAKER   speaker,
    float *   x,
    float *   y,
    FMOD_BOOL *  active
);
```

Parameters

`speaker`

The selected speaker of interest to return the x and y position.

`x`

Address of a variable that receives the 2D X position relative to the listener. Optional. Specify 0 or NULL to ignore.

`y`

Address of a variable that receives the 2D Y position relative to the listener. Optional. Specify 0 or NULL to ignore.

`active`
Address of a variable that receives the active state of a speaker.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

See the [System::set3DSpeakerPosition](#) for more information on speaker positioning.

**See Also**

- [System::set3DSpeakerPosition](#)
- **FMOD_SPEAKERMODE**
- **FMOD_SPEAKER**
Firelight Technologies FMOD Ex
**System::getAdvancedSettings**

Retrieves the advanced settings value set for the system object.

**C++ Syntax**

```cpp
FMOD_RESULT System::getAdvancedSettings(
    FMOD_ADVANCEDSETTINGS * settings
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetAdvancedSettings(
    FMOD_SYSTEM * system,
    FMOD_ADVANCEDSETTINGS * settings
);
```

**Parameters**

*settings*

Address of a variable to receive the contents of the `FMOD_ADVANCEDSETTINGS` structure specified by the user.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- `FMOD_ADVANCEDSETTINGS`
- `System::setAdvancedSettings`
Firelight Technologies FMOD Ex
System::getCDROMDriveName

Gets information on the selected cdrom drive.

C++ Syntax

FMOD_RESULT System::getCDROMDriveName(
    int drive,
    char * drivename,
    int drivenamelen,
    char * scsiname,
    int scsinamelen,
    char * devicename,
    int devicenamelen
);

C Syntax

FMOD_RESULT FMOD_System_GetCDROMDriveName(
    FMOD_SYSTEM * system,
    int drive,
    char * drivename,
    int drivenamelen,
    char * scsiname,
    int scsinamelen,
    char * devicename,
    int devicenamelen
);

Parameters

drive

The enumerated number of the CDROM drive to query. 0 based.

drivename

Address of a variable that receives the name of the drive letter or name depending on the operating system.

drivenamelen
Length in bytes of the target buffer to receive the string.

`scsiname`

Address of a variable that receives the SCSI address of the drive. This could also be used to pass to `System::createSound`, or just used for information purposes.

`scsinameLength`

Length in bytes of the target buffer to receive the string.

`devicename`

Address of a variable that receives the name of the physical device. This is usually a string defined by the manufacturer. It also contains the drive's vendor ID, product ID and version number.

`devicenameLength`

Length in bytes of the target buffer to receive the string.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Enumerate CDROM drives by finding out how many there are with `System::getNumCDROMDrives`.

**See Also**

- `System::getNumCDROMDrives`
• **System::createSound**
Firelight Technologies FMOD Ex
System::getCPUUsage

Retrieves in percent of CPU time - the amount of cpu usage that FMOD is taking for streaming/mixing and System::update combined.

C++ Syntax

```cpp
FMOD_RESULT System::getCPUUsage(
    float * dsp,
    float * stream,
    float * geometry,
    float * update,
    float * total
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetCPUUsage(
    FMOD_SYSTEM * system,
    float * dsp,
    float * stream,
    float * geometry,
    float * update,
    float * total
);
```

Parameters

**dsp**

Address of a variable that receives the current dsp mixing engine cpu usage. Result will be from 0 to 100.0f. Optional. Specify 0 or NULL to ignore.

**stream**

Address of a variable that receives the current streaming engine cpu usage. Result will be from 0 to 100.0f. Optional. Specify 0 or NULL to ignore.

**geometry**

Address of a variable that receives the current geometry engine cpu usage.
Result will be from 0 to 100.0f. Optional. Specify 0 or NULL to ignore.

update

Address of a variable that receives the current System::update cpu usage. Result will be from 0 to 100.0f. Optional. Specify 0 or NULL to ignore.

total

Address of a variable that receives the current total cpu usage. Result will be from 0 to 100.0f. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This value is slightly smoothed to provide more stable readout (and to round off spikes that occur due to multitasking/operating system issues).

NOTE! On ps3 and xbox360, the dsp and stream figures are NOT main cpu/main thread usage. On PS3 this is the percentage of SPU being used. On Xbox 360 it is the percentage of a hardware thread being used which is on a totally different CPU than the main one.
Do not be alarmed if the usage for these platforms reaches over 50%, this is normal and should be ignored if you are playing a lot of compressed sounds and are using effects. The only value on the main cpu / main thread to take note of here that will impact your framerate is the update value, and this is typically very low (ie less than 1%).

See Also
- **System::update**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
System::getChannel

Retrieves a handle to a channel by ID.

C++ Syntax

```cpp
FMOD_RESULT System::getChannel(
    int channelid,
    FMOD::Channel ** channel
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetChannel(
    FMOD_SYSTEM * system,
    int channelid,
    FMOD_CHANNEL ** channel
);
```

Parameters

channelid

Index in the FMOD channel pool. Specify a channel number from 0 to the 'maxchannels' value specified in System::init minus 1.

channel

Address of a variable that receives a pointer to the requested channel.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Remarks

This function is mainly for getting handles to existing (playing) channels and setting their attributes.

See Also

- System::playSound
- System::init
Firelight Technologies FMOD Ex
System::getChannelsPlaying

Retrieves the number of currently playing channels.

**C++ Syntax**

```cpp
FMOD_RESULT System::getChannelsPlaying(
    int * channels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetChannelsPlaying(
    FMOD_SYSTEM * system,
    int * channels
);
```

**Parameters**

*channels*

Address of a variable that receives the number of currently playing channels.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Firelight Technologies FMOD Ex
System::getDSPBufferSize

Retrieves the buffer size settings for the FMOD software mixing engine.

C++ Syntax

```cpp
FMOD_RESULT System::getDSPBufferSize(
    unsigned int * bufferlength,
    int * numbuffers
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetDSPBufferSize(
    FMOD_SYSTEM * system,
    unsigned int * bufferlength,
    int * numbuffers
);
```

Parameters

*bufferlength*

Address of a variable that receives the mixer engine block size in samples. Default = 1024. (milliseconds = 1024 at 48khz = 1024 / 48000 * 1000 = 10.66ms). This means the mixer updates every 21.3ms. Optional. Specify 0 or NULL to ignore.

*numbuffers*

Address of a variable that receives the mixer engine number of buffers used. Default = 4. To get the total buffersize multiply the bufferlength by the numbuffers value. By default this would be 4*1024. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See documentation on `System::setDSPBufferSize` for more information about these values.

**See Also**

- `System::setDSPBufferSize`
Firelight Technologies FMOD Ex
System::getDSPClock

Return the current 64bit DSP clock value which counts up by the number of samples per second in the software mixer, every second. I.e if the default sample rate is 48khz, the DSP clock increments by 48000 per second.

C++ Syntax

```cpp
FMOD_RESULT System::getDSPClock(
    unsigned int * hi,
    unsigned int * lo
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetDSPClock(
    FMOD_SYSTEM * system,
    unsigned int * hi,
    unsigned int * lo
);
```

Parameters

`hi`

The most significant 32bits of the 64bit DSP clock value.

`lo`

The least significant 32bits of the 64bit DSP clock value.

Return Values

If the function succeeds then the return value is `FMOD_OK`
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Use result with **Channel::setDelay** to play a sound on an exact tick in the future. Use **Channel::getDelay** after playing a sound to work out what DSP clock value a sound started on.

Use FMOD_64BIT_ADD or FMOD_64BIT_SUB helper macros from fmod.h to add a hi/lo combination together and cope with wraparound.

See Also

- **Channel::setDelay**
- **Channel::getDelay**
Firelight Technologies FMOD Ex
System::getDSPHead

Returns a pointer to the head DSP unit of the DSP network. This unit is the closest unit to the soundcard and all sound data comes through this unit.

C++ Syntax

```cpp
FMOD_RESULT System::getDSPHead(
    FMOD::DSP ** dsp
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetDSPHead(
    FMOD_SYSTEM * system,
    FMOD_DSP ** dsp
);
```

Parameters

`dsp`

Address of a variable that receives the pointer to the head DSP unit.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Use this unit if you wish to connect custom DSP units to the output or filter the global mix by inserting filter units between this one and the incoming channel mixer unit.
Read the tutorial on DSP if you wish to know more about this. It is not recommended using this if you do not understand how the FMOD Ex DSP network is connected.

See Also

- Channel::getDSPHead
- ChannelGroup::getDSPHead
Firelight Technologies FMOD Ex
System::getDriver

Returns the currently selected driver number. Drivers are enumerated when selecting a driver with System::setDriver or other driver related functions such as System::getNumDrivers or System::getDriverInfo

C++ Syntax

```cpp
FMOD_RESULT System::getDriver(
    int * driver
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetDriver(
    FMOD_SYSTEM * system,
    int * driver
);
```

Parameters

`driver`

Address of a variable that receives the currently selected driver ID. 0 = primary or main sound device as selected by the operating system settings.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- System::setDriver
- `System::getNumDrivers`
- `System::getDriverInfo`
Firelight Technologies FMOD Ex
System::getDriverCaps

Returns information on capabilities of the current output mode for the selected sound device.

C++ Syntax

```cpp
FMOD_RESULT System::getDriverCaps(
    int id,
    FMOD_CAPS * caps,
    int * controlpaneloutputrate,
    FMOD_SPEAKERMODE * controlpanelspeakermode
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetDriverCaps(
    FMOD_SYSTEM * system,
    int id,
    FMOD_CAPS * caps,
    int * controlpaneloutputrate,
    FMOD_SPEAKERMODE * controlpanelspeakermode
);
```

Parameters

$id$

Enumerated driver ID. This must be in a valid range delimited by `System::getNumDrivers`.

$caps$

Address of a variable that receives the capabilities of the device. Optional. Specify 0 or NULL to ignore.

$controlpaneloutputrate$

Address of a variable that receives the frequency set by the operating system. Optional. Specify 0 or NULL to ignore.
controlpanelspeakermode

Address of a variable that receives the speaker mode set by the operating system control panel. Use this to pass to System::setSpeakerMode if you want to set up FMOD's software mixing engine to match. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

**Remarks**

This function cannot be called after FMOD is already activated with System::init. It must be called before System::init, or after System::close.

**See Also**

- FMOD_CAPS
- System::init
- System::close
- System::getNumDrivers
- System::getHardwareChannels
- System::setSpeakerMode
- Channel::setFrequency
Firelight Technologies FMOD Ex
System::getDriverInfo

Retrieves identification information about a sound device specified by its index, and specific to the output mode set with System::setOutput.

C++ Syntax

```cpp
FMOD_RESULT System::getDriverInfo(
    int id,
    char * name,
    int namelen,
    FMOD_GUID * guid
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetDriverInfo(
    FMOD_SYSTEM * system,
    int id,
    char * name,
    int namelen,
    FMOD_GUID * guid
);
```

Parameters

$id$

Index of the sound driver device. The total number of devices can be found with System::getNumDrivers.

$name$

Address of a variable that receives the name of the device. Optional. Specify 0 or NULL to ignore.

$namelen$

Length in bytes of the target buffer to receive the string. Required if name parameter is not NULL.
guid

Address of a variable that receives the GUID that uniquely identifies the device. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- System::getNumDrivers
- System::setOutput
Firelight Technologies FMOD Ex
System::getDriverInfoW

Retrieves identification information about a sound device specified by its index, and specific to the output mode set with System::setOutput.

C++ Syntax

```cpp
FMOD_RESULT System::getDriverInfoW(
    int id,
    short * name,
    int namelen,
    FMOD_GUID * guid
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetDriverInfoW(
    FMOD_SYSTEM * system,
    int id,
    short * name,
    int namelen,
    FMOD_GUID * guid
);
```

Parameters

id

Index of the sound driver device. The total number of devices can be found with System::getNumDrivers.

name

Address of a variable that receives the name of the device in wide chars. Optional. Specify 0 or NULL to ignore.

namelen

Length in characters of the target buffer to receive the string. Required if name parameter is not NULL.
**guid**

Address of a variable that receives the GUID that uniquely identifies the device. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [System::getNumDrivers](#)
- [System::setOutput](#)
Firelight Technologies FMOD Ex
System::getGeometryOcclusion

Calculates geometry occlusion between a listener and a sound source.

C++ Syntax

```cpp
FMOD_RESULT System::getGeometryOcclusion(
    const FMOD_VECTOR * listener,
    const FMOD_VECTOR * source,
    float * direct,
    float * reverb
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetGeometryOcclusion(
    FMOD_SYSTEM * system,
    const FMOD_VECTOR * listener,
    const FMOD_VECTOR * source,
    float * direct,
    float * reverb
);
```

Parameters

**listener**

The listener position.

**source**

The source position.

**direct**

Optional. Specify 0 to ignore. Address of a variable to receive the direct occlusion value.

**reverb**

Optional. Specify 0 to ignore. Address of a variable to receive the reverb
occlusion value.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

If single sided polygons have been created, it is important to get the source and listener positions round the right way, as the occlusion from point A to point B may not be the same as the occlusion from point B to point A.

**See Also**

- [System::createGeometry](#)
Firelight Technologies FMOD Ex
System::getGeometrySettings

Retrieves the maximum world size for the geometry engine.

C++ Syntax

```cpp
FMOD_RESULT System::getGeometrySettings(
    float * maxworldsize
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetGeometrySettings(
    FMOD_SYSTEM * system,
    float * maxworldsize
);
```

Parameters

`maxworldsize`

Pointer to a float to receive the maximum world size.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::setGeometrySettings
Firelight Technologies FMOD Ex
**System::getHardwareChannels**

Returns the number of available hardware mixed channels.

**C++ Syntax**

```cpp
FMOD_RESULT System::getHardwareChannels(  
    int * numhardwarechannels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetHardwareChannels(  
    FMOD_SYSTEM * system,  
    int * numhardwarechannels
);
```

**Parameters**

* numhardwarechannels

Address of a variable that receives the number of available hardware mixed channels. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.  
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

NOTE: If this is called before **System::init**, you will receive the maximum possible hardware mixed channels, calling after **System::init** will give you the number actually allocated.
See Also

- System::init
- System::setHardwareChannels
Firelight Technologies FMOD Ex
**System::getMasterChannelGroup**

Retrieves a handle to the internal master channel group. This is the default channel group that all channels play on.

This channel group can be used to do things like set the master volume for all playing sounds. See the ChannelGroup API for more functionality.

**C++ Syntax**

```cpp
FMOD_RESULT System::getMasterChannelGroup(
    FMOD::ChannelGroup ** channelgroup);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetMasterChannelGroup(
    FMOD_SYSTEM * system,
    FMOD_CHANNELGROUP ** channelgroup);
```

**Parameters**

*channelgroup*

Address of a variable that receives a pointer to the master System object channel group.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.

If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**
• `System::createChannelGroup`
• `ChannelGroup::setVolume`
• `ChannelGroup::getVolume`
Firelight Technologies FMOD Ex
**System::getMasterSoundGroup**

Retrieves the default sound group, where all sounds are placed when they are created.

**C++ Syntax**

```cpp
FMOD_RESULT System::getMasterSoundGroup(
    FMOD::SoundGroup ** soundgroup
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetMasterSoundGroup(
    FMOD_SYSTEM * system,
    FMOD_SOUNDDATA ** soundgroup
);
```

**Parameters**

`soundgroup`

Address of a pointer to a SoundGroup object to receive the master sound group.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

If a user based soundgroup is deleted/released, the sounds will be put back into this sound group.
See Also

- \texttt{SoundGroup::release}
- \texttt{SoundGroup::getSystemObject}
- \texttt{SoundGroup::setMaxAudible}
- \texttt{SoundGroup::getMaxAudible}
- \texttt{SoundGroup::getName}
- \texttt{SoundGroup::getNumSounds}
- \texttt{SoundGroup::getSound}
- \texttt{SoundGroup::getNumPlaying}
- \texttt{SoundGroup::setUserData}
- \texttt{SoundGroup::getUserData}
Firelight Technologies FMOD Ex
System::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT System::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetMemoryInfo(
    FMOD_SYSTEM * system,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

event_memorybits

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

memoryused

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated
**FMOD_MEMORY_USAGE_DETAILS** structure to be filled with detailed
memory usage information about this object.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the
**FMOD_RESULT** enumeration.

**Remarks**

Every public FMOD class has a getMemoryInfo function which can be used to
get detailed information on what memory resources are associated with the
object in question. The getMemoryInfo function can be used in two different
ways:

- Use "memorybits" and "event_memorybits" to specify what memory usage
  you'd like to query and receive one number back in "memoryused".
- Provide an **FMOD_MEMORY_USAGE_DETAILS** structure for FMOD to fill
  with memory usage values for all types of memory usage.

You can use the **FMOD_MEMBITS_xxx** and **FMOD_EVENT_MEMBITS_xxx**
defines to get FMOD to add up the memory usage numbers you're interested in:

```cpp
FMOD::ChannelGroup *channelgroup;
unsigned int usedvalue;

(create channelgroup here...)

// By specifying **FMOD_MEMBITS_DSPI** and **FMOD_MEMBITS_CHANNELGROUP** headers
// only counts **FMOD_MEMBITS_DSPI** and **FMOD_MEMBITS_CHANNELGROUP** types
```
result = channelgroup->getMemoryInfo(FMOD_MEMBITS_DSP | FMOD_MEM
if (result != FMOD_OK)
{
(handle error...)
}

printf("This FMOD::ChannelGroup is currently using %d bytes for DS

Alternatively, the FMOD_MEMORY_USAGE_DETAILS structure can be used to find out the memory usage of each type within a object :

FMOD::ChannelGroup *channelgroup;
FMOD_MEMORY_USAGE_DETAILS memused_details;
unsigned int channelgroupused;

(create channelgroup here...)

// By specifying a "FMOD_MEMORY_USAGE_DETAILS" struct here, we're with memory usage values for all types of memory associated wit
result = channelgroup->getMemoryInfo(0, 0, 0, &memused_details);
if (result != FMOD_OK)
{
(handle error...)
}

channelgroupused = memused_details.dsp + memused_details.channelgroup;

printf("This FMOD::ChannelGroup is currently using %d bytes for DS

See Also

- **FMOD_MEMBITS**
- **FMOD_EVENT_MEMBITS**
- **FMOD_MEMORY_USAGE_DETAILS**
Firelight Technologies FMOD Ex
**System::getNetworkProxy**

Retrieves the URL of the proxy server used in internet streaming.

**C++ Syntax**

```cpp
FMOD_RESULT System::getNetworkProxy(
    char * proxy,
    int proxylen
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetNetworkProxy(
    FMOD_SYSTEM * system,
    char * proxy,
    int proxylen
);
```

**Parameters**

*proxy*

Address of a variable that receives the proxy server URL.

*proxylen*

Size of the buffer in bytes to receive the string.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**
- `System::setNetworkProxy`
Firelight Technologies FMOD Ex
System::getNetworkTimeout

Retrieve the timeout value for network streams

C++ Syntax

```cpp
FMOD_RESULT System::getNetworkTimeout(
    int * timeout
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetNetworkTimeout(
    FMOD_SYSTEM * system,
    int * timeout
);
```

Parameters

`timeout`

The timeout value in ms.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Firelight Technologies FMOD Ex
System::getNumCDROMDrives

Retrieves the number of available CDROM drives on the user's machine.

C++ Syntax

```cpp
FMODRESULT System::getNumCDROMDrives(
    int * numdrives
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetNumCDROMDrives(
    FMOD_SYSTEM * system,
    int * numdrives
);
```

Parameters

`numdrives`

Address of a variable that receives the number of CDROM drives.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::getCDROMDriveName
Firelight Technologies FMOD Ex
System::getNumDrivers

Retrieves the number of soundcard devices on the machine, specific to the output mode set with System::setOutput.

C++ Syntax

```cpp
FMOD_RESULT System::getNumDrivers(
    int * numdrivers
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetNumDrivers(
    FMOD_SYSTEM * system, 
    int * numdrivers
);
```

Parameters

`numdrivers`

Address of a variable that receives the number of output drivers.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

If System::setOutput is not called it will return the number of drivers available for the default output type. Use this for enumerating sound devices. Use System::getDriverInfo to get the device's name.
See Also

- System::getDriver
- System::getDriverInfo
- System::setOutput
- System::getOutput
Firelight Technologies FMOD Ex
**System::getNumPlugins**

Retrieves the number of available plugins loaded into FMOD at the current time.

**C++ Syntax**

```cpp
FMOD_RESULT System::getNumPlugins(
    FMOD_PLUGINTYPE plugintype,
    int * numplugins
);```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetNumPlugins(
    FMOD_SYSTEM * system,
    FMOD_PLUGINTYPE plugintype,
    int * numplugins
);```

**Parameters**

*plugintype*

The type of plugin type such as `FMOD_PLUGINTYPE_OUTPUT`, `FMOD_PLUGINTYPE_CODEC` or `FMOD_PLUGINTYPE_DSP`.

*numplugins*

Address of a variable that receives the number of available plugins for the selected type.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- **FMOD_PLUGINTYPE**
- **System::getPluginHandle**
Firelight Technologies FMOD Ex
System::getOutput

Retrieves the current output system FMOD is using to address the hardware.

C++ Syntax

```cpp
FMOD_RESULT System::getOutput(
    FMOD_OUTPUTTYPE * output
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetOutput(
    FMOD_SYSTEM * system,
    FMOD_OUTPUTTYPE * output
);
```

Parameters

`output`

Address of a variable that receives the current output type.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `FMOD_OUTPUTTYPE`
Firelight Technologies FMOD Ex
System::getOutputByPlugin

Returns the currently selected output as an id in the list of output plugins.

C++ Syntax

```cpp
FMOD_RESULT System::getOutputByPlugin(
    unsigned int * handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetOutputByPlugin(
    FMOD_SYSTEM * system,
    unsigned int * handle
);
```

Parameters

`handle`

Handle to a pre-existing output plugin.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [System::getNumPlugins](#)
- [System::setOutputByPlugin](#)
- [System::setOutput](#)
Firelight Technologies FMOD Ex
System::getOutputHandle

Retrieves a pointer to the system level output device module. This means a pointer to a DirectX "LPDIRECTSOUND", or a WINMM handle, or with something like with FMOD_OUTPUTTYPE_NOSOUND output, the handle will be null or 0.

C++ Syntax

```cpp
FMOD_RESULT System::getOutputHandle(
    void ** handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetOutputHandle(
    FMOD_SYSTEM * system,
    void ** handle
);
```

Parameters

`handle`

Address of a variable that receives the handle to the output mode's native hardware API object (see remarks).

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks
Must be called after System::init.
Cast the resulting pointer depending on what output system pointer you are after.

FMOD_OUTPUTTYPE_DSOUND Pointer to type DIRECTSOUND is returned.
FMOD_OUTPUTTYPE_WINMM Pointer to type HWAEOUT is returned.
FMOD_OUTPUTTYPE_WASAPI Pointer to type IAudioRenderClient is returned.
FMOD_OUTPUTTYPE_ASIO Pointer to type AsioDrivers is returned.
FMOD_OUTPUTTYPE_OSS File handle is returned, (cast to int).
FMOD_OUTPUTTYPE_ALSA Pointer to type snd_pcm_t is returned.
FMOD_OUTPUTTYPE_ASOUND Pointer to type snd_pcm_t is returned.
FMOD_OUTPUTTYPE_ESD Handle of type int is returned, as returned by so_esd_open_sound (cast to int).
FMOD_OUTPUTTYPE_COREAUDIO Handle of type AudioUnit is returned.
FMOD_OUTPUTTYPE_PS3 NULL / 0 is returned.
FMOD_OUTPUTTYPE_XBOX360 Pointer to type IXAudio2 is returned.
FMOD_OUTPUTTYPE_PSP NULL / 0 is returned.
FMOD_OUTPUTTYPE_WII NULL / 0 is returned.
FMOD_OUTPUTTYPE_NOSOUND NULL / 0 is returned.
FMOD_OUTPUTTYPE_WAVWRITER NULL / 0 is returned.

See Also

- FMOD_OUTPUTTYPE
- System::setOutput
- System::init
Firelight Technologies FMOD Ex
System::getPluginHandle

Retrieves the handle of a plugin based on its type and relative index. Use System::getNumPlugins to enumerate plugins.

C++ Syntax

```cpp
FMOD_RESULT System::getPluginHandle(
    FMOD_PLUGINTYPE     plugintype,
    int                 index,
    unsigned int       * handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetPluginHandle(
    FMOD_SYSTEM        * system,
    FMOD_PLUGINTYPE    plugintype,
    int               index,
    unsigned int     * handle
);
```

Parameters

plugintype

The type of plugin type such as FMOD_PLUGINTYPE_OUTPUT, FMOD_PLUGINTYPE_CODEC or FMOD_PLUGINTYPE_DSP.

index

The relative index for the type of plugin.

handle

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- [System::getNumPlugins](#)
Firelight Technologies FMOD Ex
System::getPluginInfo

Retrieves information to display for the selected plugin.

C++ Syntax

```cpp
FMOD_RESULT System::getPluginInfo(
    unsigned int handle,
    FMOD_PLUGINTYPE * plugintype,
    char * name,
    int namelen,
    unsigned int * version
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetPluginInfo(
    FMOD_SYSTEM * system,
    unsigned int handle,
    FMOD_PLUGINTYPE * plugintype,
    char * name,
    int namelen,
    unsigned int * version
);
```

Parameters

`handle`

Handle to a pre-existing plugin.

`plugintype`

Address of a variable that receives the type of the plugin, `FMOD_PLUGINTYPE_OUTPUT`, `FMOD_PLUGINTYPE_CODEC` or `FMOD_PLUGINTYPE_DSP`.

`name`

Address of a variable that receives the name of the plugin.
name

Length in bytes of the target buffer to receive the string.

version

Version number set by the plugin.

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [System::getNumPlugins](#)
Firelight Technologies FMOD Ex
System::getRecordDriverCaps

Returns information on capabilities of the current output mode for the selected recording sound device.

C++ Syntax

```cpp
FMOD_RESULT System::getRecordDriverCaps(
    int id,
    FMOD_CAPS * caps,
    int * minfrequency,
    int * maxfrequency
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetRecordDriverCaps(
    FMOD_SYSTEM * system,
    int id,
    FMOD_CAPS * caps,
    int * minfrequency,
    int * maxfrequency
);
```

Parameters

id

Enumerated driver ID. This must be in a valid range delimited by System::getRecordNumDrivers.

caps

Address of a variable that receives the capabilities of the device. Optional. Specify 0 or NULL to ignore.

minfrequency

Address of a variable that receives the minimum frequency allowed for sounds used with recording. Optional. Specify 0 or NULL to ignore.
maxfrequency

Address of a variable that receives the maximum frequency allowed for sounds used with recording. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- FMOD_CAPS
- System::getRecordNumDrivers
Firelight Technologies FMOD Ex
System::getRecordDriverInfo

Retrieves identification information about a sound device specified by its index, and specific to the output mode set with System::setOutput.

C++ Syntax

```cpp
FMOD_RESULT System::getRecordDriverInfo(
    int id,
    char * name,
    int namelen,
    FMOD_GUID * guid
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetRecordDriverInfo(
    FMOD_SYSTEM * system,
    int id,
    char * name,
    int namelen,
    FMOD_GUID * guid
);
```

Parameters

id

Index into the enumerated list of record devices up to the value returned by System::getRecordNumDrivers.

name

Address of a variable that receives the name of the recording device. Optional. Specify 0 or NULL to ignore.

namelen

Length in bytes of the target buffer to receive the string. Required if name parameter is not NULL.
**guid**

Address of a variable that receives the GUID that uniquely identifies the device. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [System::setOutput](#)
- [System::getRecordNumDrivers](#)
Firelight Technologies FMOD Ex
System::getRecordDriverInfoW

Retrieves identification information about a sound device specified by its index, and specific to the output mode set with System::setOutput.

C++ Syntax

```cpp
FMOD_RESULT System::getRecordDriverInfoW(
    int id,
    short * name,
    int namelen,
    FMOD_GUID * guid
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetRecordDriverInfoW(
    FMOD_SYSTEM * system,
    int id,
    short * name,
    int namelen,
    FMOD_GUID * guid
);
```

Parameters

* id

Index into the enumerated list of record devices up to the value returned by System::getRecordNumDrivers.

* name

Address of a variable that receives the name of the recording device in wide chars. Optional. Specify 0 or NULL to ignore.

* namelen

Length in bytes of the target buffer to receive the string. Required if name parameter is not NULL.
guid

Address of a variable that receives the GUID that uniquely identifies the device. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `System::setOutput`
- `System::getRecordNumDrivers`
Firelight Technologies FMOD Ex
System::getRecordNumDrivers

Retrieves the number of recording devices available for this output mode. Use this to enumerate all recording devices possible so that the user can select one.

C++ Syntax

```cpp
FMOD_RESULT System::getRecordNumDrivers(
    int * numdrivers
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetRecordNumDrivers(
    FMOD_SYSTEM * system,
    int * numdrivers
);
```

Parameters

`numdrivers`

Address of a variable that receives the number of recording drivers available for this output mode.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::GetRecordDriverInfo
Firelight Technologies FMOD Ex
System::getRecordPosition

Retrieves the current recording position of the record buffer in PCM samples.

C++ Syntax

```cpp
FMOD_RESULT System::getRecordPosition(
    int id,
    unsigned int * position
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetRecordPosition(
    FMOD_SYSTEM * system,
    int id,
    unsigned int * position
);
```

Parameters

id

Enumerated driver ID. This must be in a valid range delimited by System::getRecordNumDrivers.

position

Address of a variable to receive the current recording position in PCM samples.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Firelight Technologies FMOD Ex
System::getReverbAmbientProperties

Retrieves the default reverb environment for the virtual reverb system.

C++ Syntax

```cpp
FMOD_RESULT System::getReverbAmbientProperties(
    FMOD_REVERB_PROPERTIES * prop
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetReverbAmbientProperties(
    FMOD_SYSTEM * system,
    FMOD_REVERB_PROPERTIES * prop
);
```

Parameters

`prop`

Address of a pointer to a `FMOD_REVERB_PROPERTIES` to receive the settings for the current ambient reverb setting.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

By default the ambient reverb is set to 'off'. This is the same as `FMOD_REVERB_PRESET_OFF`. 
See Also

- **FMOD_REVERB_PROPERTIES**
- **System::setReverbAmbientProperties**
- **System::createReverb**
Firelight Technologies FMOD Ex
System::getReverbProperties

Retrieves the current reverb environment for the specified reverb instance. You must specify the 'Instance' value (usually 0 unless you are using multiple reverbs) before calling this function.

C++ Syntax

```cpp
FMOD_RESULT System::getReverbProperties(
    FMOD_REVERB_PROPERTIES * prop
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetReverbProperties(
    FMOD_SYSTEM * system,
    FMOD_REVERB_PROPERTIES * prop
);
```

Parameters

`prop`

Address of a variable that receives the current reverb environment description. Make sure the 'Instance' value is specified.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

**Note!** It is important to specify the 'Instance' value in the
**FMOD_REVERB_PROPERTIES** structure correctly, otherwise you will get an **FMOD_ERR_REVERB_INSTANCE** error.

**See Also**

- **FMOD_REVERB_PROPERTIES**
- **System::setReverbProperties**
- **Channel::setReverbProperties**
- **Channel::getReverbProperties**
Firelight Technologies FMOD Ex
**System::getSoftwareChannels**

Retrieves the maximum number of software mixed channels possible. Software mixed voices are used by sounds loaded with **FMOD_SOFTWARE**.

**C++ Syntax**

```cpp
FMOD_RESULT System::getSoftwareChannels(
    int * numsoftwarechannels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetSoftwareChannels(
    FMOD_SYSTEM * system,
    int * numsoftwarechannels
);
```

**Parameters**

`numsoftwarechannels`

Address of a variable that receives the current maximum number of software voices available. Default = 32.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- **System::setSoftwareChannels**
Firelight Technologies FMOD Ex
System::getSoftwareFormat

Retrieves the output format for the software mixer.

**C++ Syntax**

```cpp
FMOD_RESULT System::getSoftwareFormat(
    int * samplerate,
    FMOD_SOUND_FORMAT * format,
    int * numoutputchannels,
    int * maxinputchannels,
    FMOD_DSP_RESAMPLER * resamplemethod,
    int * bits
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetSoftwareFormat(
    FMOD_SYSTEM * system,
    int * samplerate,
    FMOD_SOUND_FORMAT * format,
    int * numoutputchannels,
    int * maxinputchannels,
    FMOD_DSP_RESAMPLER * resamplemethod,
    int * bits
);
```

**Parameters**

*samplerate*

Address of a variable that receives the mixer's output rate. Optional. Specify 0 or NULL to ignore.

*format*

Address of a variable that receives the mixer's output format. Optional. Specify 0 or NULL to ignore.

*numoutputchannels*
Address of a variable that receives the number of output channels to initialize the mixer to, for example 1 = mono, 2 = stereo. 8 is the maximum for soundcards that can handle it. Optional. Specify 0 or NULL to ignore.

maxinputchannels

Address of a variable that receives the maximum channel depth on sounds that are loadable or creatable. Specify 0 or NULL to ignore.

resamplemethod

Address of a variable that receives the current resampling (frequency conversion) method for software mixed sounds. Specify 0 or NULL to ignore.

bits

Address of a variable that receives the number of bits per sample. Useful for byte-&gt;sample conversions. for example FMOD_SOUND_FORMAT_PCM16 is 16. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Note that the settings returned here may differ from the settings provided by the user with System::setSoftwareFormat. This is because the driver may have changed it because it will not initialize to anything else.

See Also

- System::setSoftwareFormat
- FMOD_SOUND_FORMAT
- FMOD_DSP_RESAMPLER
Firelight Technologies FMOD Ex
System::getSoundRAM

Retrieves the amount of dedicated sound ram available if the platform supports it. Most platforms use main ram to store audio data, so this function usually isn't necessary.

C++ Syntax

```cpp
FMOD_RESULT System::getSoundRAM(
    int * currentallocated,
    int * maxallocated,
    int * total
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetSoundRAM(
    FMOD_SYSTEM * system,
    int * currentallocated,
    int * maxallocated,
    int * total
);
```

Parameters

currentallocated

Address of a variable that receives the currently allocated sound ram memory at time of call. Optional. Specify 0 or NULL to ignore.

maxallocated

Address of a variable that receives the maximum allocated sound ram memory since System::init. Optional. Specify 0 or NULL to ignore.

total

Address of a variable that receives the total amount of sound ram available on this device.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [Memory_GetStats](#)
Firelight Technologies FMOD Ex
System::getSpeakerMode

Retrieves the current speaker mode.

C++ Syntax

```cpp
FMOD_RESULT System::getSpeakerMode(
    FMOD_SPEAKERMODE * speakermode
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetSpeakerMode(
    FMOD_SYSTEM * system,
    FMOD_SPEAKERMODE * speakermode
);
```

Parameters

`speakermode`

Address of a variable that receives the current speaker mode.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::setSpeakerMode
- FMOD_SPEAKERMODE
Firelight Technologies FMOD Ex
System::getSpectrum

Retrieves the spectrum from the currently playing output signal.

C++ Syntax

```cpp
FMOD_RESULT System::getSpectrum(
    float * spectrumarray,
    int numvalues,
    int channeloffset,
    FMOD_DSP_FFT_WINDOW windowtype
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetSpectrum(
    FMOD_SYSTEM * system,
    float * spectrumarray,
    int numvalues,
    int channeloffset,
    FMOD_DSP_FFT_WINDOW windowtype
);
```

Parameters

`spectrumarray`

Address of a variable that receives the spectrum data. This is an array of floating point values. Data will range is 0.0 to 1.0. Decibels = 10.0f * (float)log10(val) * 2.0f; See remarks for what the data represents.

`numvalues`

Size of array in floating point values being passed to the function. Must be a power of 2. (ie 128/256/512 etc). Min = 64. Max = 8192.

`channeloffset`

Channel of the signal to analyze. If the signal is multichannel (such as a stereo output), then this value represents which channel to analyze. On a stereo signal 0
"Pre-FFT" window method. This filters the PCM data before entering the spectrum analyzer to reduce transient frequency error for more accurate results. See FMOD_DSP_FFT_WINDOW for different types of fft window techniques possible and for a more detailed explanation.

**Return Values**

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

**Remarks**

The larger the numvalues, the more CPU the FFT will take. Choose the right value to trade off between accuracy / speed.

The larger the numvalues, the more 'lag' the spectrum will seem to inherit. This is because the FFT window size stretches the analysis back in time to what was already played. For example if the window size happened to be 44100 and the output rate was 44100 it would be analyzing the past second of data, and giving you the average spectrum over that time period.

If you are not displaying the result in dB, then the data may seem smaller than it should be. To display it you may want to normalize the data - that is, find the maximum value in the resulting spectrum, and scale all values in the array by 1 / max. (ie if the max was 0.5f, then it would become 1).

To get the spectrum for both channels of a stereo signal, call this function twice, once with channeloffset = 0, and again with channeloffset = 1. Then add the spectrums together and divide by 2 to get the average spectrum for both channels.

What the data represents.

To work out what each entry in the array represents, use this formula
entry_hz = (output_rate / 2) / numvalues

The array represents amplitudes of each frequency band from 0hz to the nyquist rate. The nyquist rate is equal to the output rate divided by 2.
For example when FMOD is set to 44100hz output, the range of represented frequencies will be 0hz to 22049hz, a total of 22050hz represented.
If in the same example, 1024 was passed to this function as the numvalues, each entry's contribution would be as follows.

entry_hz = (44100 / 2) / 1024
entry_hz = 21.53 hz

Note: This function only displays data for sounds playing that were created with FMOD_SOFTWARE. FMOD_HARDWARE based sounds are played using the sound card driver and are not accessable.

See Also

- FMOD_DSP_FFT_WINDOW
- Channel::getSpectrum
- ChannelGroup::getSpectrum
- System::getWaveData
Firelight Technologies FMOD Ex
System::getStreamBufferSize

Returns the current internal buffersize settings for streamable sounds.

C++ Syntax

```cpp
FMOD_RESULT System::getStreamBufferSize(
    unsigned int * filebuffersize,
    FMOD_TIMEUNIT * filebuffersizetype
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetStreamBufferSize(
    FMOD_SYSTEM * system,
    unsigned int * filebuffersize,
    FMOD_TIMEUNIT * filebuffersizetype
);
```

Parameters

filebuffersize

Address of a variable that receives the current stream file buffer size setting. Default is 16384 (FMOD_TIMEUNIT_RAWBYTES). Optional. Specify 0 or NULL to ignore.

filebuffersizetype

Address of a variable that receives the type of unit for the current stream file buffer size setting. Can be FMOD_TIMEUNIT_MS, FMOD_TIMEUNIT_PCM, FMOD_TIMEUNIT_PCMBYTES or FMOD_TIMEUNIT_RAWBYTES. Default is FMOD_TIMEUNIT_RAWBYTES. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `FMOD_TIMEUNIT`
- `System::setStreamBufferSize`
Firelight Technologies FMOD Ex
System::getUserData

Retrieves the user value that that was set by calling the System::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT System::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetUserData(
    FMOD_SYSTEM * system,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the System::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- System::setUserData
Firelight Technologies FMOD Ex
System::getVersion

Returns the current version of FMOD Ex being used.

C++ Syntax

```cpp
FMOD_RESULT System::getVersion(
    unsigned int * version
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetVersion(
    FMOD_SYSTEM * system,
    unsigned int * version
);
```

Parameters

`version`

Address of a variable that receives the current FMOD Ex version.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

The version is a 32bit hexadecimal value formatted as 16:8:8, with the upper 16bits being the major version, the middle 8bits being the minor version and the bottom 8bits being the development version. For example a value of 00040106h is equal to 4.01.06.
See Also

- System::init
Firelight Technologies FMOD Ex
System::getWaveData

Retrieves a pointer to a block of PCM data that represents the currently playing audio mix. This function is useful for a very easy way to plot an oscilloscope.

C++ Syntax

```cpp
FMOD_RESULT System::getWaveData(
    float * wavearray,
    int numvalues,
    int channeloffset
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetWaveData(
    FMOD_SYSTEM * system,
    float * wavearray,
    int numvalues,
    int channeloffset
);
```

Parameters

`wavearray`

Address of a variable that receives the currently playing waveform data. This is an array of floating point values.

`numvalues`

Number of floats to write to the array. Maximum value = 16384.

`channeloffset`

Offset into multichannel data. For mono output use 0. Stereo output will use 0 = left, 1 = right. More than stereo output - use the appropriate index.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This is the actual resampled, filtered and volume scaled data of the final output, at the time this function is called.

Do not use this function to try and display the whole waveform of the sound, as this is more of a 'snapshot' of the current waveform at the time it is called, and could return the same data if it is called very quickly in succession. See the DSP API to capture a continual stream of wave data as it plays, or see `Sound::lock` / `Sound::unlock` if you want to simply display the waveform of a sound.

This function allows retrieval of left and right data for a stereo sound individually. To combine them into one signal, simply add the entries of each separate buffer together and then divide them by 2.

**Note:** This function only displays data for sounds playing that were created with `FMOD_SOFTWARE`. `FMOD_HARDWARE` based sounds are played using the sound card driver and are not accessable.

See Also

- `System::getSpectrum`
- `Channel::getWaveData`
- `ChannelGroup::getWaveData`
- `Sound::lock`
- `Sound::unlock`
Firelight Technologies FMOD Ex
System::init

Initializes the system object, and the sound device. This has to be called at the start of the user's program.

You must create a system object with FMOD::System_create.

C++ Syntax

```cpp
FMOD_RESULT System::init(
    int maxchannels,
    FMOD_INITFLAGS flags,
    void * extradriverrdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_Init(
    FMOD_SYSTEM * system,
    int maxchannels,
    FMOD_INITFLAGS flags,
    void * extradriverrdata
);
```

Parameters

maxchannels

The maximum number of channels to be used in FMOD. They are also called 'virtual channels' as you can play as many of these as you want, even if you only have a small number of hardware or software voices. See remarks for more.

flags

See FMOD_INITFLAGS. This can be a selection of flags bitwise OR'ed together to change the behaviour of FMOD at initialization time.

extradriverrdata

Driver specific data that can be passed to the output plugin. For example the filename for the wav writer plugin. See FMOD_OUTPUTTYPE for what each
output mode might take here. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

**Virtual channels.**
These types of voices are the ones you work with using the FMOD::Channel API.
The advantage of virtual channels are, unlike older versions of FMOD, you can now play as many sounds as you like without fear of ever running out of voices, or playsound failing.
You can also avoid 'channel stealing' if you specify enough virtual voices.

As an example, you can play 1000 sounds at once, even on a 32 channel soundcard.
FMOD will only play the most important/closest/loudest (determined by volume/distance/geometry and priority settings) voices, and the other 968 voices will be virtualized without expense to the CPU. The voice's cursor positions are updated.

When the priority of sounds change or emulated sounds get louder than audible ones, they will swap the actual voice resource over (ie hardware or software buffer) and play the voice from its correct position in time as it should be heard. What this means is you can play all 1000 sounds, if they are scattered around the game world, and as you move around the world you will hear the closest or most important 32, and they will automatically swap in and out as you move.

Currently the maximum channel limit is 4093.

**See Also**
- FMOD_INITFLAGS
- System::close
- System Create
- FMOD_OUTPUTTYPE
Firelight Technologies FMOD Ex
System::isRecording

Retrieves the state of the FMOD recording API, ie if it is currently recording or not.

**C++ Syntax**

```cpp
FMOD_RESULT System::isRecording(
    int id,
    bool * recording
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_IsRecording(
    FMOD SYSTEM * system,
    int id,
    FMOD BOOL * recording
);
```

**Parameters**

*id*

Enumerated driver ID. This must be in a valid range delimited by System::getRecordNumDrivers.

*recording*

Address of a variable to receive the current recording state. True or non zero if the FMOD recording api is currently in the middle of recording, false or zero if the recording api is stopped / not recording.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Recording can be started with `System::recordStart`.

See Also

- `System::recordStart`
- `System::recordStop`
Firelight Technologies FMOD Ex
System::loadGeometry

Creates a geometry object from a block of memory which contains pre-saved geometry data, saved by Geometry::save.

C++ Syntax

```cpp
FMOD_RESULT System::loadGeometry(
    const void * data,
    int datasize,
    FMOD::Geometry ** geometry
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_LoadGeometry(
    FMOD_SYSTEM * system,
    const void * data,
    int datasize,
    FMOD_GEOMETRY ** geometry
);
```

Parameters

data
Address of data containing pre-saved geometry data.

datasize
Size of geometry data block in bytes.

geometry
Address of a variable to receive a newly created FMOD::Geometry object.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- Geometry::save
- System::createGeometry
Firelight Technologies FMOD Ex
System::loadPlugin

Loads an FMOD plugin. This could be a DSP, file format or output plugin.

C++ Syntax

```cpp
FMOD_RESULT System::loadPlugin(
    const char * filename,
    unsigned int * handle,
    unsigned int priority
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_LoadPlugin(
    FMOD_SYSTEM * system,
    const char * filename,
    unsigned int * handle,
    unsigned int priority
);
```

Parameters

filename

Filename of the plugin to be loaded.

handle

Pointer to an unsigned int to receive the plugin handle, for later use.

priority

(FMOD_PLUGINTYPE_CODEC only) Priority of the codec compared to other codecs. 0 = most important. higher numbers = less importance.

Return Values
If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

Once the plugin is loaded, it can be enumerated and used.
For file format plugins, FMOD will automatically try to use them when **System::createSound** is used.
For DSP plugins, you can enumerate them with **System::getNumPlugins**, **System::getPluginHandle** and **System::getPluginInfo**.
Plugins can be created for FMOD by the user. See the relevant section in the documentation on creating plugins.
The format of the plugin is dependant on the operating system.
On Win32 and Win64 the .dll format is used
On Linux, the .so format is used.
On Macintosh, the .shlib format is used

The codecs internal to FMOD have the following priorities.

<table>
<thead>
<tr>
<th>Codec</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>100</td>
</tr>
<tr>
<td>CDDA</td>
<td>200</td>
</tr>
<tr>
<td>FSB</td>
<td>300</td>
</tr>
<tr>
<td>DSP</td>
<td>400</td>
</tr>
<tr>
<td>VAG</td>
<td>500</td>
</tr>
<tr>
<td>Wav</td>
<td>600</td>
</tr>
<tr>
<td>AT3</td>
<td>700</td>
</tr>
<tr>
<td>OggVorbis</td>
<td>800</td>
</tr>
<tr>
<td>Tremor</td>
<td>900</td>
</tr>
<tr>
<td>AIFF</td>
<td>1000</td>
</tr>
<tr>
<td>FLAC</td>
<td>1100</td>
</tr>
<tr>
<td>MOD</td>
<td>1200</td>
</tr>
<tr>
<td>S3M</td>
<td>1300</td>
</tr>
<tr>
<td>XM</td>
<td>1400</td>
</tr>
<tr>
<td>IT</td>
<td>1500</td>
</tr>
<tr>
<td>MIDI</td>
<td>1600</td>
</tr>
<tr>
<td>DLS</td>
<td>1700</td>
</tr>
<tr>
<td>SF2</td>
<td>1800</td>
</tr>
<tr>
<td>ASF</td>
<td>1900</td>
</tr>
<tr>
<td>XMA</td>
<td>2000</td>
</tr>
<tr>
<td>XWMA</td>
<td>2100</td>
</tr>
</tbody>
</table>
Playlist 2200
MPEG PSP 2300
MPEG 2400
Raw 2500

**Note:** Some codecs are only for certain platforms, ie XMA is xbox 360 only.

**See Also**

- [System::setPluginPath](#)
- [System::unloadPlugin](#)
- [System::getNumPlugins](#)
- [System::getPluginHandle](#)
- [System::getPluginInfo](#)
- [System::setOutputByPlugin](#)
- [System::getOutputByPlugin](#)
- [System::createDSPByPlugin](#)
- [System::createSound](#)
Firelight Technologies FMOD Ex
System::lockDSP

Mutual exclusion function to lock the FMOD DSP engine (which runs asynchronously in another thread), so that it will not execute. If the FMOD DSP engine is already executing, this function will block until it has completed.

The function may be used to synchronize DSP network operations carried out by the user.

An example of using this function may be for when the user wants to construct a DSP sub-network, without the DSP engine executing in the background while the sub-network is still under construction.

### C++ Syntax

FMOD_RESULT System::lockDSP();

### C Syntax

FMOD_RESULT FMOD_System_LockDSP(FMOD_SYSTEM * system);

### Parameters

### Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

### Remarks

Once the user no longer needs the DSP engine locked, it must be unlocked with System::unlockDSP.
Note that the DSP engine should not be locked for a significant amount of time, otherwise inconsistency in the audio output may result. (audio
skipping/stuttering).

See Also

- System::unlockDSP
Firelight Technologies FMOD Ex
System::playDSP

Plays a DSP unit object and its input network on a particular channel.

C++ Syntax

```cpp
FMOD_RESULT System::playDSP(
    FMOD_CHANNELINDEX channelId,
    FMOD::DSP * dsp,
    bool paused,
    FMOD::Channel ** channel
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_PlayDSP(
    FMOD_SYSTEM * system,
    FMOD_CHANNELINDEX channelId,
    FMOD_DSP * dsp,
    FMOD_BOOL paused,
    FMOD_CHANNEL ** channel
);
```

Parameters

**channelid**

Use the value **FMOD_CHANNEL_FREE** to get FMOD to pick a free channel. Otherwise specify a channel number from 0 to the 'maxchannels' value specified in **System::init** minus 1.

**dsp**

Pointer to the dsp unit to play. This is opened with **System::createDSP**, **System::createDSPByType**, **System::createDSPByPlugin**.

**paused**

True or false flag to specify whether to start the channel paused or not. Starting a channel paused allows the user to alter its attributes without it being audible, and
unpausing with `Channel::setPaused` actually starts the dsp running.

`channel`

Address of a channel handle pointer that receives the newly playing channel. If `FMOD_CHANNEL_REUSE` is used, this can contain a previously used channel handle and FMOD will re-use it to play a dsp on.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

When a dsp is played, it will use the dsp's default frequency, volume, pan, levels and priority.

A dsp defined as `FMOD_3D` will by default play at the position of the listener.

To change channel attributes before the dsp is audible, start the channel paused by setting the paused flag to true, and calling the relevant channel based functions. Following that, unpause the channel with `Channel::setPaused`.

If `FMOD_CHANNEL_FREE` is used as the channel index, it will pick an arbitrary free channel and use channel management. (As described below). If `FMOD_CHANNEL_REUSE` is used as the channel index, FMOD Ex will re-use the channel handle that is passed in as the 'channel' parameter. If NULL or 0 is passed in as the channel handle it will use the same logic as `FMOD_CHANNEL_FREE` and pick an arbitrary channel.

Channels are reference counted. If a channel is stolen by the FMOD priority system, then the handle to the stolen voice becomes invalid, and Channel based commands will not affect the new channel playing in its place. If all channels are currently full playing a dsp or sound, FMOD will steal a
channel with the lowest priority dsp or sound. If more channels are playing than are currently available on the soundcard/sound device or software mixer, then FMOD will 'virtualize' the channel. This type of channel is not heard, but it is updated as if it was playing. When its priority becomes high enough or another sound stops that was using a real hardware/software channel, it will start playing from where it should be. This technique saves CPU time (thousands of sounds can be played at once without actually being mixed or taking up resources), and also removes the need for the user to manage voices themselves.

An example of virtual channel usage is a dungeon with 100 torches burning, all with a looping crackling sound, but with a soundcard that only supports 32 hardware voices. If the 3D positions and priorities for each torch are set correctly, FMOD will play all 100 sounds without any 'out of channels' errors, and swap the real voices in and out according to which torches are closest in 3D space.

Priority for virtual channels can be changed in the sound's defaults, or at runtime with Channel::setPriority.

See Also

- FMOD_CHANNELINDEX
- System::createDSP
- System::createDSPByType
- System::createDSPByPlugin
- Channel::setPaused
- Channel::setPriority
- DSP::setDefaults
- System::init
Firelight Technologies FMOD Ex
System::playSound

Plays a sound object on a particular channel.

C++ Syntax

```cpp
FMOD_RESULT System::playSound(
    FMOD_CHANNELINDEX channelid,
    FMOD::Sound * sound,
    bool paused,
    FMOD::Channel ** channel
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_PlaySound(
    FMOD_SYSTEM * system,
    FMOD_CHANNELINDEX channelid,
    FMOD_SOUND * sound,
    FMOD_BOOL paused,
    FMOD_CHANNEL ** channel
);
```

Parameters

channelid

Use the value `FMOD_CHANNEL_FREE` to get FMOD to pick a free channel. Otherwise specify a channel number from 0 to the 'maxchannels' value specified in `System::init` minus 1.

sound

Pointer to the sound to play. This is opened with `System::createSound`.

paused

True or false flag to specify whether to start the channel paused or not. Starting a channel paused allows the user to alter its attributes without it being audible, and unpausing with `Channel::setPaused` actually starts the sound.
channel

Address of a channel handle pointer that receives the newly playing channel. If `FMOD_CHANNEL_REUSE` is used, this can contain a previously used channel handle and FMOD will re-use it to play a sound on.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

When a sound is played, it will use the sound's default frequency, volume, pan, levels and priority.

A sound defined as `FMOD_3D` will by default play at the position of the listener.

To change channel attributes before the sound is audible, start the channel paused by setting the paused flag to true, and calling the relevant channel based functions. Following that, unpause the channel with `Channel::setPaused`.

If `FMOD_CHANNEL_FREE` is used as the channel index, it will pick an arbitrary free channel and use channel management. (As described below). If `FMOD_CHANNEL_REUSE` is used as the channel index, FMOD Ex will re-use the channel handle that is passed in as the 'channel' parameter. If NULL or 0 is passed in as the channel handle it will use the same logic as `FMOD_CHANNEL_FREE` and pick an arbitrary channel.

Channels are reference counted. If a channel is stolen by the FMOD priority system, then the handle to the stolen voice becomes invalid, and Channel based commands will not affect the new sound playing in its place.
If all channels are currently full playing a sound, FMOD will steal a channel with the lowest priority sound.
If more channels are playing than are currently available on the soundcard/sound
device or software mixer, then FMOD will 'virtualize' the channel. This type of channel is not heard, but it is updated as if it was playing. When its priority becomes high enough or another sound stops that was using a real hardware/software channel, it will start playing from where it should be. This technique saves CPU time (thousands of sounds can be played at once without actually being mixed or taking up resources), and also removes the need for the user to manage voices themselves.

An example of virtual channel usage is a dungeon with 100 torches burning, all with a looping crackling sound, but with a soundcard that only supports 32 hardware voices. If the 3D positions and priorities for each torch are set correctly, FMOD will play all 100 sounds without any 'out of channels' errors, and swap the real voices in and out according to which torches are closest in 3D space.

Priority for virtual channels can be changed in the sound's defaults, or at runtime with Channel::setPriority.

See Also

- FMOD_CHANNELINDEX
- System::createSound
- Channel::setPaused
- Channel::setPriority
- Sound::setDefaults
- Sound::setVariations
- System::init
Firelight Technologies FMOD Ex
System::recordStart

Starts the recording engine recording to the specified recording sound.

C++ Syntax

```cpp
FMOD_RESULT System::recordStart(
    int id,
    FMOD::Sound * sound,
    bool loop
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_RecordStart(
    FMOD_SYSTEM * system,
    int id,
    FMOD_SOUND * sound,
    FMOD_BOOL loop
);
```

Parameters

`id`

Enumerated driver ID. This must be in a valid range delimited by System::getRecordNumDrivers.

`sound`

User created sound for the user to record to.

`loop`

Boolean flag to tell the recording engine whether to continue recording to the provided sound from the start again, after it has reached the end. If this is set to true the data will be continually be overwritten once every loop. See remarks.
**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- **System::recordStop**
Firelight Technologies FMOD Ex
System::recordStop

Stops the recording engine from recording to the specified recording sound.

C++ Syntax

```cpp
FMOD_RESULT System::recordStop(
  int id
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_RecordStop(
  FMOD_SYSTEM * system,
  int id
);
```

Parameters

```
id
```

Enumerated driver ID. This must be in a valid range delimited by System::getRecordNumDrivers.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::recordStart
Firelight Technologies FMOD Ex
System::registerCodec

Creates an in memory file format codec to be used by FMOD by passing in a codec description structure.
Once this is created, FMOD will use it to open user defined file formats.

C++ Syntax

```cpp
FMOD_RESULT System::registerCodec(
    FMOD_CODEC_DESCRIPTION * description,
    unsigned int * handle,
    unsigned int priority
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_RegisterCodec(
    FMOD_SYSTEM * system,
    FMOD_CODEC_DESCRIPTION * description,
    unsigned int * handle,
    unsigned int priority
);
```

Parameters

`description`
Address of a `FMOD_CODEC_DESCRIPTION` structure, containing information about the codec.

`handle`
Address of a variable to receive the plugin handle of the newly-registered codec.

`priority`
Priority of the codec compared to other codecs. 0 = most important. higher numbers = less importance.
**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

The codecs internal to FMOD have the following priorities.

<table>
<thead>
<tr>
<th>Codec</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>100</td>
</tr>
<tr>
<td>CDDA</td>
<td>200</td>
</tr>
<tr>
<td>FSB</td>
<td>300</td>
</tr>
<tr>
<td>DSP</td>
<td>400</td>
</tr>
<tr>
<td>VAG</td>
<td>500</td>
</tr>
<tr>
<td>Wav</td>
<td>600</td>
</tr>
<tr>
<td>AT3</td>
<td>700</td>
</tr>
<tr>
<td>OggVorbis</td>
<td>800</td>
</tr>
<tr>
<td>Tremor</td>
<td>900</td>
</tr>
<tr>
<td>AIFF</td>
<td>1000</td>
</tr>
<tr>
<td>FLAC</td>
<td>1100</td>
</tr>
<tr>
<td>MOD</td>
<td>1200</td>
</tr>
<tr>
<td>S3M</td>
<td>1300</td>
</tr>
<tr>
<td>XM</td>
<td>1400</td>
</tr>
<tr>
<td>IT</td>
<td>1500</td>
</tr>
<tr>
<td>MIDI</td>
<td>1600</td>
</tr>
<tr>
<td>DLS</td>
<td>1700</td>
</tr>
<tr>
<td>SF2</td>
<td>1800</td>
</tr>
<tr>
<td>ASF</td>
<td>1900</td>
</tr>
<tr>
<td>XMA</td>
<td>2000</td>
</tr>
<tr>
<td>XWMA</td>
<td>2100</td>
</tr>
<tr>
<td>Playlist</td>
<td>2200</td>
</tr>
<tr>
<td>MPEGPSP</td>
<td>2300</td>
</tr>
<tr>
<td>MPEG</td>
<td>2400</td>
</tr>
<tr>
<td>Raw</td>
<td>2500</td>
</tr>
</tbody>
</table>

**Note:** Some codecs are only for certain platforms, i.e., XMA is Xbox 360 only.

**See Also**
• **FMOD_CODEC_DESCRIPTION**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
System::registerDSP

Register a user-defined DSP effect for use with the System. This function allows you to register statically-linked DSP effects. Once registered, you can create instances of the DSP effect by using System::createDSPByPlugin.

C++ Syntax

```cpp
FMOD_RESULT System::registerDSP(
    FMOD_DSP_DESCRIPTION * description,
    unsigned int * handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_RegisterDSP(
    FMOD_SYSTEM * system,
    FMOD_DSP_DESCRIPTION * description,
    unsigned int * handle
);
```

Parameters

description

Address of an FMOD_DSP_DESCRIPTION structure, containing information about the DSP effect.

handle

Address of a variable to receive the plugin handle of the newly-registered DSP effect.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
See Also

- **FMOD_DSP_DESCRIPTION**
- **System::createDSPByPlugin**
- **System::getNumPlugins**
- **System::getPluginInfo**
Firelight Technologies FMOD Ex
System::release

Closes and frees a system object and its resources.

C++ Syntax

FMOD_RESULT System::release();

C Syntax

FMOD_RESULT FMOD_System_Release(FMOD_SYSTEM * system);

Parameters

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function also calls System::close, so calling close before this function is not necessary.

See Also

- System_Create
- System::init
- System::close
Firelight Technologies FMOD Ex
System::set3DListenerAttributes

This updates the position, velocity and orientation of the specified 3D sound listener.

C++ Syntax

```cpp
FMOD_RESULT System::set3DListenerAttributes(
    int listener,
    const FMOD_VECTOR * pos,
    const FMOD_VECTOR * vel,
    const FMOD_VECTOR * forward,
    const FMOD_VECTOR * up
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_Set3DListenerAttributes(
    FMOD_SYSTEM * system,
    int listener,
    const FMOD_VECTOR * pos,
    const FMOD_VECTOR * vel,
    const FMOD_VECTOR * forward,
    const FMOD_VECTOR * up
);
```

Parameters

**listener**

Listener ID in a multi-listener environment. Specify 0 if there is only 1 listener.

**pos**

The position of the listener in world space, measured in distance units. You can specify 0 or NULL to not update the position.

**vel**

The velocity of the listener measured in distance units **per second**. You can specify 0 or NULL to not update the velocity of the listener.
*forward*

The forwards orientation of the listener. This vector must be of unit length and perpendicular to the up vector. You can specify 0 or NULL to not update the forwards orientation of the listener.

*up*

The upwards orientation of the listener. This vector must be of unit length and perpendicular to the forwards vector. You can specify 0 or NULL to not update the upwards orientation of the listener.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.  
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

By default, FMOD uses a left-handed co-ordinate system. This means +X is right, +Y is up, and +Z is forwards.  
To change this to a right-handed coordinate system, use **FMOD_INIT_3D_RIGHTHANDED**. This means +X is right, +Y is up, and +Z is backwards or towards you.

To map to another coordinate system, flip/negate and exchange these values.

Orientation vectors are expected to be of UNIT length. This means the magnitude of the vector should be 1.0.

A 'distance unit' is specified by **System::set3DSettings**. By default this is set to meters which is a distance scale of 1.0.

Always remember to use **units per second**, not units per frame as this is a common mistake and will make the doppler effect sound wrong.
For example, Do not just use (pos - lastpos) from the last frame's data for velocity, as this is not correct. You need to time compensate it so it is given in units per **second**.
You could alter your pos - lastpos calculation to something like this.

\[
\text{vel} = \frac{\text{(pos-lastpos)}}{\text{time_taken_since_last_frame_in_seconds}}.
\]

I.e. at 60fps the formula would look like this \(\text{vel} = \frac{\text{(pos-lastpos)}}{0.0166667}\).

**See Also**

- **System::get3DListenerAttributes**
- **FMOD_INITFLAGS**
- **System::set3DSettings**
- **System::get3DSettings**
- **FMOD_VECTOR**
Firelight Technologies FMOD Ex
System::set3DNumListeners

Sets the number of 3D 'listeners' in the 3D sound scene. This function is useful mainly for split-screen game purposes.

**C++ Syntax**

```cpp
FMOD_RESULT System::set3DNumListeners(
    int numlisteners
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_Set3DNumListeners(
    FMOD_SYSTEM * system,
    int numlisteners
);
```

**Parameters**

`numlisteners`

Number of listeners in the scene. Valid values are from 1-4 inclusive. Default = 1.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

If the number of listeners is set to more than 1, then panning and doppler are turned off. All sound effects will be mono.
FMOD uses a 'closest sound to the listener' method to determine what should be heard in this case.

See Also

- System::get3DNumListeners
- System::set3DLlistenerAttributes
Firelight Technologies FMOD Ex
System::set3DRolloffCallback

When FMOD wants to calculate 3d volume for a channel, this callback can be used to override the internal volume calculation based on distance.

C++ Syntax

```cpp
FMOD_RESULT System::set3DRolloffCallback(
            FMOD_3D_ROLLOFFCALLBACK callback
        );
```

C Syntax

```c
FMOD_RESULT FMOD_System_Set3DRolloffCallback(
            FMOD_SYSTEM * system,
            FMOD_3D_ROLLOFFCALLBACK callback
        );
```

Parameters

`callback`

Pointer to a C function of type `FMOD_3D_ROLLOFFCALLBACK`, that is used to override the FMOD volume calculation. Default is 0 or NULL. Setting the callback to null will return 3d calculation back to FMOD.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function overrides `FMOD_3D_INVERSEROLLOFF`,
FMOD_3D_LINEARROLLOFF, FMOD_3D_LINEARSQUAREROLLOFF, FMOD_3D_CUSTOMROLLOFF. To allow FMOD to calculate the 3d volume again, use 0 or NULL as the callback.

NOTE: When using the event system, call Channel::getUserData from your FMOD_3D_ROLLOFFCALLBACK to get the event instance handle of the event that spawned the channel in question.

See Also

- FMOD_3D_ROLLOFFCALLBACK
- System::set3DListenerAttributes
- System::get3DListenerAttributes
- Channel::getUserData
Firelight Technologies FMOD Ex
System::set3DSettings

Sets the global doppler scale, distance factor and log rolloff scale for all 3D sound in FMOD.

C++ Syntax

```cpp
FMOD_RESULT System::set3DSettings(
    float dopplerscale,
    float distancefactor,
    float rolloffscale
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_Set3DSettings(
    FMOD_SYSTEM * system,
    float dopplerscale,
    float distancefactor,
    float rolloffscale
);
```

Parameters

*dopplerscale*

Scaling factor for doppler shift. Default = 1.0.

*distancefactor*

Relative distance factor to FMOD's units. Default = 1.0. (1.0 = 1 metre).

*rolloffscale*

Scaling factor for 3D sound rolloff or attenuation for `FMOD_3D_INVERSEROLLOFF` based sounds only (which is the default type). Default = 1.0.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

The **doppler scale** is a general scaling factor for how much the pitch varies due to doppler shifting in 3D sound. Doppler is the pitch bending effect when a sound comes towards the listener or moves away from it, much like the effect you hear when a train goes past you with its horn sounding. With "dopplerscale" you can exaggerate or diminish the effect. FMOD's effective speed of sound at a doppler factor of 1.0 is 340 m/s.

The **distance factor** is the FMOD 3D engine relative distance factor, compared to 1.0 meters. Another way to put it is that it equates to "how many units per meter does your engine have". For example, if you are using feet then "scale" would equal 3.28.

**Note!** This only affects doppler! If you keep your min/max distance, custom rolloff curves and positions in scale relative to each other the volume rolloff will not change. If you set this, the mindistance of a sound will automatically set itself to this value when it is created in case the user forgets to set the mindistance to match the new distancefactor.

The **rolloff scale** sets the global attenuation rolloff factor for **FMOD_3D_INVERSEROLLOFF** based sounds only (which is the default). Volume for a sound set to **FMOD_3D_INVERSEROLLOFF** will scale at mindistance / distance. This gives an inverse attenuation of volume as the source gets further away (or closer). Setting this value makes the sound drop off faster or slower. The higher the value, the faster volume will attenuate, and conversely the lower the value, the slower it will attenuate. For example a rolloff factor of 1 will simulate the real world, where as a value of 2 will make sounds attenuate 2 times quicker.

**Note!** "rolloffscale" has no effect when using **FMOD_3D_LINEARROLLOFF**, **FMOD_3D_LINEARQUASEROLLOFF** or **FMOD_3D_CUSTOMROLLOFF**.
See Also

- System::get3DSettings
- Sound::set3DMinMaxDistance
- Sound::get3DMinMaxDistance
- Channel::set3DAttributes
- Channel::get3DAttributes
System::set3DSpeakerPosition

This function allows the user to specify the position of their actual physical speaker to account for non standard setups.
It also allows the user to disable speakers from 3D consideration in a game.
The function is for describing the 'real world' speaker placement to provide a more natural panning solution for 3d sound. Graphical configuration screens in an application could draw icons for speaker placement that the user could position at their will.

C++ Syntax

```c++
FMOD_RESULT System::set3DSpeakerPosition(
    FMOD_SPEAKER speaker,
    float x,
    float y,
    bool active
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_Set3DSpeakerPosition(
    FMOD_SYSTEM * system,
    FMOD_SPEAKER speaker,
    float x,
    float y,
    FMOD_BOOL active
);
```

Parameters

`speaker`

The selected speaker of interest to position.

`x`

The 2D X offset in relation to the listening position. For example -1.0 would mean the speaker is on the left, and +1.0 would mean the speaker is on the right. 0.0 is the speaker is in the middle.
The 2D Y offset in relation to the listening position. For example -1.0 would mean the speaker is behind the listener, and +1 would mean the speaker is in front of the listener.

**active**

Enables or disables speaker from 3D consideration. Useful for disabling center speaker for vocals for example, or the LFE. x and y can be anything in this case.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

**Note!** This only affects software mixed 3d sounds, created with **FMOD_SOFTWARE** and **FMOD_3D**.

A typical 7.1 setup would look like this.

```c
system->set3DSpeakerPosition(FMOD_SPEAKER_FRONT_LEFT, -1.0f, 1.0f, true);
system->set3DSpeakerPosition(FMOD_SPEAKER_FRONT_RIGHT, 1.0f, 1.0f, true);
system->set3DSpeakerPosition(FMOD_SPEAKER_FRONT_CENTER, 0.0f, 1.0f, true);
system->set3DSpeakerPosition(FMOD_SPEAKER_LOW_FREQUENCY, 0.0f, 0.0f, true);
system->set3DSpeakerPosition(FMOD_SPEAKER_BACK_LEFT, -1.0f, -1.0f, true);
system->set3DSpeakerPosition(FMOD_SPEAKER_BACK_RIGHT, 1.0f, -1.0f, true);
system->set3DSpeakerPosition(FMOD_SPEAKER_SIDE_LEFT, -1.0f, 0.0f, true);
system->set3DSpeakerPosition(FMOD_SPEAKER_SIDE_RIGHT, 1.0f, 0.0f, true);
```

A typical stereo setup would look like this.

```c
system->set3DSpeakerPosition(FMOD_SPEAKER_FRONT_LEFT, -1.0f, 0.0f, true);
system->set3DSpeakerPosition(FMOD_SPEAKER_FRONT_RIGHT, 1.0f, 0.0f, true);
```

You could use this function to make sounds in front of your come out of
different physical speakers. If you specified for example that **FMOD_SPEAKER_SIDE_RIGHT** was in front of you at <0.0, 1.0> and you organized the other speakers accordingly the 3d audio would come out of the side right speaker when it was in front instead of the default which is only to the side.

This function is also useful if speakers are not 'perfectly symmetrical'. For example if the center speaker was closer to the front left than the front right, this function could be used to position that center speaker accordingly and FMOD would skew the panning appropriately to make it sound correct again.

The 2d coordinates used are only used to generate angle information. Size / distance does not matter in FMOD's implementation because it is not FMOD's job to attenuate or amplify the signal based on speaker distance. If it amplified the signal in the digital domain the audio could clip/become distorted. It is better to use the amplifier's analogue level capabilities to balance speaker volumes. Calling `System::setSpeakerMode` overrides these values, so this function must be called after this.

**See Also**

- `System::get3DSpeakerPosition`
- `System::setSpeakerMode`
- **FMOD_SPEAKERMODE**
- **FMOD_SPEAKER**
Firelight Technologies FMOD Ex
System::setAdvancedSettings

Sets advanced features like configuring memory and cpu usage for FMOD_CREATECOMPRESSESAMPLE usage.

C++ Syntax

```cpp
FMOD_RESULT System::setAdvancedSettings(
    FMOD_ADVANCEDSETTINGS * settings
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetAdvancedSettings(
    FMOD_SYSTEM * system,
    FMOD_ADVANCEDSETTINGS * settings
);
```

Parameters

`settings`

Pointer to FMOD_ADVANCEDSETTINGS structure.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- FMOD_ADVANCEDSETTINGS
- System::getAdvancedSettings
- FMOD_MODE
Firelight Technologies FMOD Ex
**System::setCallback**

Sets a system callback to catch various fatal or informational events.

**C++ Syntax**

```cpp
FMOD_RESULT System::setCallback(
    FMOD_SYSTEM_CALLBACK callback
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_SetCallback(
    FMOD_SYSTEM * system,
    FMOD_SYSTEM_CALLBACK callback
);
```

**Parameters**

`callback`

Pointer to a callback to receive the event callback when it happens.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

System callbacks are not asynchronous and are bound by the latency caused by the rate the user calls the update command.

Callbacks are stdcall. Use `F_CALLBACK` inbetween your return type and
Example:

```c
FMOD_RESULT F_CALLBACK systemcallback(FMOD_SYSTEM *system, FMOD_SYSTEM_CALLBACKTYPE type) {
    FMOD::System *sys = (FMOD::System *)system;

    switch (type) {
    case FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED:
        {
            int numdrivers;

            printf("NOTE : FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED\n");
            sys->getNumDrivers(&numdrivers);

            printf("Numdevices = %d\n", numdrivers);
            break;
        }

    case FMOD_SYSTEM_CALLBACKTYPE_MEMORYALLOCATIONFAILED:
        {
            printf("ERROR : FMOD_SYSTEM_CALLBACKTYPE_MEMORYALLOCATIONFAILED\n");
            printf("%s\n", commanddata1);
            printf("%d bytes\n", commanddata2);
            break;
        }

    case FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED:
        {
            printf("NOTE : FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED\n");
            printf("Thread ID = %d\n", (int)commanddata1);
            printf("Thread Name = %s\n", (char *)commanddata2);
            break;
        }

    case FMOD_SYSTEM_CALLBACKTYPE_BADDSPCONNECTION:
        {
            FMOD::DSP *source = (FMOD::DSP *)commanddata1;
            FMOD::DSP *dest = (FMOD::DSP *)commanddata2;

            printf("ERROR : FMOD_SYSTEM_CALLBACKTYPE_BADDSPCONNECTION\n");
            if (source) {
                char name[256];
                
```
source->getInfo(name, 0, 0, 0, 0);
printf("SOURCE = %s\n", name);
}
if (dest)
{
    char name[256];
    dest->getInfo(name, 0, 0, 0, 0);
    printf("DEST = %s\n", name);
    break;
}
}

return FMOD_OK;


See Also

- System::update
- FMOD_SYSTEM_CALLBACK
- FMOD_SYSTEM_CALLBACKTYPE

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
System::setDSPBufferSize

Sets the FMOD internal mixing buffer size. This function is used if you need to control mixer latency or granularity. Smaller buffersizes lead to smaller latency, but can lead to stuttering/skipping/instable sound on slower machines or soundcards with bad drivers.

C++ Syntax

```cpp
FMOD_RESULT System::setDSPBufferSize(
    unsigned int bufferlength,
    int numbuffers
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetDSPBufferSize(
    FMOD_SYSTEM * system,
    unsigned int bufferlength,
    int numbuffers
);
```

Parameters

`bufferlength`

The mixer engine block size in samples. Use this to adjust mixer update granularity. Default = 1024. (milliseconds = 1024 at 48khz = 1024 / 48000 * 1000 = 21.33ms). This means the mixer updates every 21.33ms.

`numbuffers`

The mixer engine number of buffers used. Use this to adjust mixer latency. Default = 4. To get the total buffersize multiply the bufferlength by the numbuffers value. By default this would be 4*1024.

Return Values
If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

The FMOD software mixer mixes to a ringbuffer. The size of this ringbuffer is determined here. It mixes a block of sound data every 'bufferlength' number of samples, and there are 'numbuffers' number of these blocks that make up the entire ringbuffer. Adjusting these values can lead to extremely low latency performance (smaller values), or greater stability in sound output (larger values).

Warning! The 'buffersize' is generally best left alone. Making the granularity smaller will just increase CPU usage (cache misses and DSP network overhead). Making it larger affects how often you hear commands update such as volume/pitch/pan changes. Anything above 20ms will be noticable and sound parameter changes will be obvious instead of smooth.

FMOD chooses the most optimal size by default for best stability, depending on the output type, and if the drivers are emulated or not (for example DirectSound is emulated using waveOut on NT). It is not recommended changing this value unless you really need to. You may get worse performance than the default settings chosen by FMOD.

To convert from milliseconds to 'samples', simply multiply the value in milliseconds by the sample rate of the output (ie 48000 if that is what it is set to), then divide by 1000.

The values in milliseconds and average latency expected from the settings can be calculated using the following code.

```c
FMOD_RESULT result;
unsigned int blocksize;
int numbblocks;
float ms;
```
result = system->getDSPBufferSize(&blocksize, &numblocks);
result = system->getSoftwareFormat(&frequency, 0, 0, 0, 0);

ms = (float)blocksize * 1000.0f / (float)frequency;

printf("Mixer blocksize = %.02f ms\n", ms);
printf("Mixer Total buffer size = %.02f ms\n", ms * numblocks);
printf("Mixer Average Latency = %.02f ms\n", ms * ((float)numblocks - 1.5f));

**Platform notes:** Some output modes (such as `FMOD_OUTPUTTYPE_ASIO`) will change the buffer size to match their own internal optimal buffer size. Use `System::getDSPBufferSize` after calling `System::init` to see if this is the case. Linux output modes will ignore numbuffers and just write the buffer size to the output every time it can. It does not use a ringbuffer. Xbox 360 defaults to 256 sample buffersize and 4 for numblocks. This gives a 5.333ms granularity with roughly a 10-15ms latency. PS3 ignores this function. Check `FMOD_PS3_EXTRADRIVERDATA` to control output latency. This function cannot be called after FMOD is already activated with `System::init`. It must be called before `System::init`, or after `System::close`.

**See Also**

- `System::getDSPBufferSize`
- `System::getSoftwareFormat`
- `System::init`
- `System::close`
System::setDriver

Selects a soundcard driver. This function is used when an output mode has enumerated more than one output device, and you need to select between them.

C++ Syntax

```cpp
FMOD_RESULT System::setDriver(
    int driver
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetDriver(
    FMOD_SYSTEM * system,
    int driver
);
```

Parameters

driver

Driver number to select. 0 = primary or main sound device as selected by the operating system settings. Use System::getNumDrivers to select a specific device.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

If this function is called after FMOD is already initialized with System::init, the
current driver will be shutdown and the newly selected driver will be initialized / started.

When switching output driver after `System::init` there are a few considerations to make:

All sounds must be created with `FMOD_SOFTWARE`, creating even one `FMOD_HARDWARE` sound will cause this function to return `FMOD_ERR_NEEDSSOFTWARE`.

The driver that you wish to change to must support the current output format, sample rate, and number of channels. If it does not, `FMOD_ERR_OUTPUT_INIT` is returned and driver state is cleared. You should now call `System::setDriver` with your original driver index to restore driver state (providing that driver is still available / connected) or make another selection.

See Also

- `System::getDriver`
- `System::getNumDrivers`
- `System::getDriverInfo`
- `System::setOutput`
- `System::init`
- `System::close`
Firelight Technologies FMOD Ex
System::setFileSystem

Specify user callbacks for FMOD's internal file manipulation functions.
If ANY of the callback functions are set to 0/ NULL, then FMOD will switch back to its own file routines.
This function is useful for replacing FMOD's file system with a game system's own file reading API.

C++ Syntax

```c++
FMOD_RESULT System::setFileSystem(
    FMOD_FILE_OPENCALLBACK    useropen,
    FMOD_FILE_CLOSECALLBACK   userclose,
    FMOD_FILE_READCALLBACK    userread,
    FMOD_FILE_SEEKCALLBACK    userseek,
    FMOD_FILE_ASYNCREADCALLBACK userasyncread,
    FMOD_FILEASYNCCANCELCALLBACK   userasynccancel,
    int    blockalign
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetFileSystem(
    FMOD_SYSTEM    *    system,
    FMOD_FILE_OPENCALLBACK    useropen,
    FMOD_FILE_CLOSECALLBACK   userclose,
    FMOD_FILE_READCALLBACK    userread,
    FMOD_FILE_SEEKCALLBACK    userseek,
    FMOD_FILE_ASYNCREADCALLBACK userasyncread,
    FMOD_FILEASYNCCANCELCALLBACK userasynccancel,
    int    blockalign
);
```

Parameters

useropen

Callback for opening a file. Specifying 0 / null will disable file callbacks.

userclose
Callback for closing a file. Specifying 0 / null will disable file callbacks.

**userread**

Callback for reading from a file. Specifying 0 / null will disable file callbacks if userasyncread is also 0 / null. User could use userasyncread instead of userread.

**userseek**

Callback for seeking within a file. Specifying 0 / null will disable file callbacks. User could use userasyncread instead of userseek.

**userasyncread**

OPTIONAL - Callback to replace 'userread' and 'userseek' that allows the user to defer file access to a later time and return immediately. FMOD will internally wait for data to appear, or in a file streaming case - stutter/starve if data is not fed to fmod in time. Set to 0 / null to get normal file callback operation.

**userasynccancel**

OPTIONAL - Callback for cancelling pending user file accesses. This will be called if a sound is released, so the user can cancel any pending file accesses. If the sound is released and a deferred read happens into a released buffer, the application will crash. This callback must be used to make sure this doesn't happen. Set to 0 / null to get normal file callback operation.

**blockalign**

Internal minimum file block alignment. FMOD will read data in at least chunks of this size if you ask it to. Specifying 0 means there is no file buffering at all (this could adversely affect streaming). Do NOT make this a large value, it is purely a setting for minimum sector size alignment to aid seeking and reading on certain media. It is not for stream buffer sizes, that is what System:::setStreamBufferSize is for. It is recommened just to pass -1. Large values just mean large memory usage with no benefit. Specify -1 to not set this value. Default = 2048.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

This has no effect on sounds loaded with **FMOD_OPENMEMORY** or **FMOD_CREATEUSER**.

This function can be used to set user file callbacks, or if required, they can be turned off by specifying 0 for all callbacks.
This function can be used purely to set the 'buffersize' parameter, and ignore the callback aspect of the function.

Warning: This function can cause unpredictable behaviour if not used properly. You must return the right values, and each command must work properly, or FMOD will not function, or it may even crash if you give it invalid data. You must also return **FMOD_ERR_FILE_EOF** from a read callback if the number of bytes read is smaller than the number of bytes requested.

FMOD's default filesystem buffers reads every 2048 bytes by default. This means every time fmod reads one byte from the API (say if it was parsing a file format), it simply mem copies the byte from the 2k memory buffer, and every time it needs to, refreshes the 2k buffer resulting in a drastic reduction in file I/O. Large reads go straight to the pointer instead of the 2k buffer if it is buffer aligned. This value can be increased or decreased by the user. A buffer of 0 means all reads go directly to the pointer specified. 2048 bytes is the size of a CD sector on most CD ISO formats so it is chosen as the default, for optimal reading speed from CD media.

**NOTE!** Do not force a cast from your function pointer to the **FMOD_FILE_xxx_CALLBACK** type! Never try to 'force' fmod to accept your function. If there is an error then find out what it is. Remember to include `F_CALLBACK` between the return type and the function name, this equates to stdcall which you must include otherwise (besides not compiling) it will cause
problems such as crashing and callbacks not being called.

**NOTE!** Your file callbacks must be thread safe. If not unexpected behaviour may occur. FMOD calls file functions from asynchronous threads, such as the streaming thread, and thread related to `FMOD_NONBLOCKING` flag.

Asynchronous file access (userasyncread/userasynccancel). For 'userasyncread' and 'userasynccancel' usage, see the 'asyncio' example in the FMOD examples folder. There is also a tutorial in the documentation. If userasyncread callback is specified - userread and userseek will not be called at all, so they can be set to 0 / null. userasyncread allows the user to return immediately before the data is ready. FMOD will either wait internally (see note below about thread safety), or poll in the streamer until data arrives. It is the user's responsibility to provide data in time in the stream case, or the stream may stutter. Data starvation can be detected with `Sound::getOpenState`.

**NOTE:** If userasyncread is processed in the main thread, then it will hang the application, because FMOD will wait internally until data is ready, and the main thread process will not be able to supply the data. For this reason the user's file access should normally be from a separate thread.

**See Also**

- `System::init`
- `System::attachFileSystem`
- `Sound::getOpenState`
- `FMOD_FILE_OPENCALLBACK`
- `FMOD_FILE_CLOSECALLBACK`
- `FMOD_FILE_READCALLBACK`
- `FMOD_FILE_SEEKCALLBACK`
- `FMOD_FILE_ASYNCREADCALLBACK`
- `FMOD_FILEASYNCCANCELCALLBACK`
Firelight Technologies FMOD Ex
System::setGeometrySettings

Sets the maximum world size for the geometry engine for performance / precision reasons.

C++ Syntax

```cpp
FMOD_RESULT System::setGeometrySettings(
    float maxworldsize
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetGeometrySettings(
    FMOD_SYSTEM * system,
    float maxworldsize
);
```

Parameters

`maxworldsize`

Maximum size of the world from the centerpoint to the edge using the same units used in other 3D functions.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Setting `maxworldsize` should be done first before creating any geometry. It can be done any time afterwards but may be slow in this case.
Objects or polygons outside the range of maxworldsize will not be handled efficiently. Conversely, if maxworldsize is excessively large, the structure may lose precision and efficiency may drop.

See Also

- System::createGeometry
- System::getGeometrySettings
Firelight Technologies FMOD Ex
System::setHardwareChannels

This function allows the user to request how many hardware mixed channels will be acquired during System::init.

C++ Syntax

```cpp
FMOD_RESULT System::setHardwareChannels(
    int numhardwarechannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetHardwareChannels(
    FMOD_SYSTEM * system,
    int numhardwarechannels
);
```

Parameters

`numhardwarechannels`

Maximum number of hardware voices to be used by FMOD. This clamps the polyphony of hardware voices to a user specified number. This could be used to limit the number of hardware voices possible at once so that it doesn't sound noisy.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks
This function cannot be called after FMOD is already activated with System::init.
It must be called before System::init, or after System::close.

See Also

- System::getHardwareChannels
- System::init
- System::close
Firelight Technologies FMOD Ex
System::setNetworkProxy

Set a proxy server to use for all subsequent internet connections.

C++ Syntax

```cpp
FMOD_RESULT System::setNetworkProxy(
    const char * proxy
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetNetworkProxy(
    FMOD_SYSTEM * system,
    const char * proxy
);
```

Parameters

proxy

The name of a proxy server in host:port format e.g. www.fmod.org:8888 (defaults to port 80 if no port is specified).

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Basic authentication is supported. To use it, this parameter must be in user:password@host:port format e.g. bob:sekrit123@www.fmod.org:8888 Set this parameter to 0 / NULL if no proxy is required.
See Also

- System::getNetworkProxy
Firelight Technologies FMOD Ex
System::setNetworkTimeout

Set the timeout for network streams.

C++ Syntax

```cpp
FMOD_RESULT System::setNetworkTimeout(
    int timeout
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetNetworkTimeout(
    FMOD_SYSTEM * system,
    int timeout
);
```

Parameters

`timeout`

The timeout value in ms.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `System::getNetworkTimeout`
Firelight Technologies FMOD Ex
System::setOutput

This function selects the output mode for the platform. This is for selecting different OS specific APIs which might have different features.

**C++ Syntax**

```cpp
FMOD_RESULT System::setOutput(
    FMOD_OUTPUTTYPE output
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_SetOutput(
    FMOD_SYSTEM * system,
    FMOD_OUTPUTTYPE output
);
```

**Parameters**

`output`

Output type to select. See type list for different output types you can select.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This function is not necessary to call. It is only if you want to specifically switch away from the default output mode for the operating system. The most optimal mode is selected by default for the operating system. For example
FMOD_OUTPUTTYPE_DSOUND is selected on all operating systems except for Windows NT, where FMOD_OUTPUTTYPE_WINMM is selected because it is lower latency / faster.

This function cannot be called after FMOD is already activated with System::init. It must be called before System::init, or after System::close.

See Also

- FMOD_OUTPUTTYPE
- System::init
- System::close

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
System::setOutputByPlugin

Selects an output type based on the enumerated list of outputs including FMOD and 3rd party output plugins.

C++ Syntax

```cpp
FMOD_RESULT System::setOutputByPlugin(
    unsigned int   handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetOutputByPlugin(
    FMOD_SYSTEM * system,
    unsigned int   handle
);
```

Parameters

handle

Handle to a pre-existing output plugin.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function cannot be called after FMOD is already activated with System::init. It must be called before System::init.
See Also

- System::getNumPlugins
- System::getOutputByPlugin
- System::setOutput
- System::init
- System::close
Firelight Technologies FMOD Ex
System::setPluginPath

Specify a base search path for plugins so they can be placed somewhere else than the directory of the main executable.

C++ Syntax

```cpp
FMOD_RESULT System::setPluginPath(
    const char * path
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetPluginPath(
    FMOD_SYSTEM * system,
    const char * path
);
```

Parameters

`path`

A character string containing a correctly formatted path to load plugins from.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

The 'plugin' version of FMOD relies on plugins, so when `System::init` is called it tries to load all FMOD registered plugins. This path is where it will attempt to load from.
See Also

- System::loadPlugin
- System::init
Firelight Technologies FMOD Ex
System::setReverbAmbientProperties

Sets a 'background' default reverb environment for the virtual reverb system. This is a reverb preset that will be morphed to if the listener is not within any virtual reverb zones.

By default the ambient reverb is set to 'off'.

C++ Syntax

```cpp
FMOD_RESULT System::setReverbAmbientProperties(
    FMOD_REVERB_PROPERTIES * prop
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetReverbAmbientProperties(
    FMOD_SYSTEM * system,
    FMOD_REVERB_PROPERTIES * prop
);
```

Parameters

**prop**

Address of a FMOD_REVERB_PROPERTIES structure containing the settings for the desired ambient reverb setting.

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks
There is one reverb DSP dedicated to providing a 3D reverb effect. This DSP's properties are a weighted sum of all the contributing virtual reverbs. The default 3d reverb properties specify the reverb properties in the 3D volumes which has no virtual reverbs defined.

Note about physical SFX Reverb DSP unit allocation. To remove the DSP unit and the associated CPU cost, make sure all 3d 3d reverb objects are released, and the ambient reverb background property is set to OFF (environment = -1). If either a 3d reverb zone is still present or the ambient setting is not off, the SFX Reverb DSP unit will remain active. Simply creating 1 reverb object, or setting the ambient state to something other than OFF, an SFX Reverb DSP unit will be created and inserted back into the DSP network.

The user may wish to avoid a sudden cutout of reverb by leaving the ambient setting active for a while. Make sure to release all objects and set the ambient reverb setting to off to get a proper cleanup.

See Also

- **FMOD_REVERB_PROPERTIES**
- **System::getReverbAmbientProperties**
- **System::createReverb**
Firelight Technologies FMOD Ex
System::setReverbProperties

Sets parameters for the global reverb environment.
  Reverb parameters can be set manually, or automatically using the pre-defined presets given in the fmod.h header.

C++ Syntax

```cpp
FMOD_RESULT System::setReverbProperties(
    const FMOD_REVERB_PROPERTIES * prop
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetReverbProperties(
    FMOD_SYSTEM * system,
    const FMOD_REVERB_PROPERTIES * prop
);
```

Parameters

`prop`

Address of an `FMOD_REVERB_PROPERTIES` structure which defines the attributes for the reverb.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Note! It is important to specify the 'Instance' value in the
FMOD_REVERB_PROPERTIES structure correctly, otherwise you will get an FMOD_ERR_REVERB_INSTANCE error.

See Also

- FMOD_REVERB_PROPERTIES
- System::getReverbProperties
- Channel::setReverbProperties
- Channel::getReverbProperties
Firelight Technologies FMOD Ex
System::setSoftwareChannels

Sets the maximum number of software mixed channels possible. Software mixed voices are used by sounds loaded with FMOD_SOFTWARE.

C++ Syntax

```cpp
FMOD_RESULT System::setSoftwareChannels(
    int numsoftwarechannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetSoftwareChannels(
    FMOD_SYSTEM * system,
    int numsoftwarechannels
);
```

Parameters

`numsoftwarechannels`

The maximum number of FMOD_SOFTWARE mixable voices to be allocated by FMOD. If you don't require software mixed voices specify 0. Default = 32.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

32 voices are allocated by default to be played simultaneously in software.
To turn off the software mixer completely including hardware resources used for
the software mixer, specify `FMOD_INIT_SOFTWARE_DISABLE` in `System::init`.

This function cannot be called after FMOD is already activated with `System::init`. It must be called before `System::init`, or after `System::close`.

See Also

- `FMOD_MODE`
- `FMOD_INITFLAGS`
- `System::init`
- `System::close`
- `System::getSoftwareChannels`
Firelight Technologies FMOD Ex
System::setSoftwareFormat

Sets the output format for the software mixer. This includes the bitdepth, sample rate and number of output channels.

Do not call this unless you explicitly want to change something. Calling this could have adverse impact on the performance and panning behaviour.

C++ Syntax

```cpp
FMOD_RESULT System::setSoftwareFormat(
    int samplerate,
    FMOD_SOUND_FORMAT format,
    int numoutputchannels,
    int maxinputchannels,
    FMOD_DSP_RESAMPLER resamplemethod
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetSoftwareFormat(
    FMOD_SYSTEM * system,
    int samplerate,
    FMOD_SOUND_FORMAT format,
    int numoutputchannels,
    int maxinputchannels,
    FMOD_DSP_RESAMPLER resamplemethod
);
```

Parameters

samplerate

The soundcard's output rate. default = 48000.

format

The soundcard's output format. default = FMOD_SOUND_FORMAT_PCM16.

numoutputchannels
The number of output channels / speakers to initialize the soundcard to. 0 = keep spekermode setting (set with System::setSpeakerMode). If anything else than 0 is specified then the spekermode will be overriden and will become FMOD_SPEAKERMODE_RAW, meaning logical speaker assignments (as defined in FMOD_SPEAKER) become ineffectve and cannot be used. Channel::setPan will also fail. Default = 2 (FMOD_SPEAKERMODE_STEREO).

maxinputchannels

Optional. Specify 0 to ignore. Default = 6. Maximum channel count in loaded/created sounds to be supported. This is here purely for memory considerations and affects how much memory is used in the software mixer when allocating matrices for panning. Do not confuse this with recording, or anything to do with how many voices you can play at once. This is purely for setting the largest type of sound you can play (ie 1 = mono, 2 = stereo, etc.). Most of the time the user will not play sounds any larger than mono or stereo, so setting this to 2 would save memory and cover most sounds that are playable.

resamplemethod

Software engine resampling method. default = FMOD_DSP_RESAMPLER_LINEAR. See FMOD_DSP_RESAMPLER for different types.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Note! The settings in this function may be overriden by the output mode. FMOD_OUTPUTTYPE_ASIO will always change the output mode to FMOD_SOUND_FORMAT_PCMFLOAT to be compatible with the output
formats selectable by the ASIO control panel. FMOD_OUTPUTTYPE_ASIO will also change the samplerate specified by the user to the one selected in the ASIO control panel. Use System::getSoftwareFormat after System::init to determine what the output has possibly changed the format to. Call it after System::init.

It is dependant on the output whether it will force a format change and override these settings or not.

If the output does not support the output mode specified System::init will fail, and you will have to try another setting.

Note! When this function is called with a output channel count greater than 0, the speaker mode is set to FMOD_SPEAKERMODE_RAW. FMOD does not know when you specify a number of output channels what type of speaker system it is connected to, so Channel::setPan or Channel::setSpeakerMix will then fail to work. Calling System::setSpeakerMode will override the output channel speaker count.

This function cannot be called after FMOD is already activated with System::init. It must be called before System::init, or after System::close.

See Also

- System::getSoftwareFormat
- System::setSpeakerMode
- System::init
- System::close
- Channel::setPan
- Channel::setSpeakerMix
- FMOD_SPEAKER
- FMOD_SPEAKERMODE
- FMOD_SOUND_FORMAT
- FMOD_DSP_RESAMPLER
Firelight Technologies FMOD Ex
System::setSpeakerMode

Sets the speaker mode in the hardware and FMOD software mixing engine.

C++ Syntax

```cpp
FMOD_RESULT System::setSpeakerMode(
    FMOD_SPEAKERMODE speakermode
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetSpeakerMode(
    FMOD_SYSTEM * system,
    FMOD_SPEAKERMODE speakermode
);
```

Parameters

`speakermode`

Speaker mode specified from the list in `FMOD_SPEAKERMODE`.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Speaker modes that are supported on each platform are as follows.

- Win32/Win64 - All.
- Linux/Linux64 - All.
- Macintosh - All.
• iOS - `FMOD_SPEAKERMODE_RAW`, `FMOD_SPEAKERMODE_MONO`, `FMOD_SPEAKERMODE_STEREO`.
• PSP - `FMOD_SPEAKERMODE_RAW`, `FMOD_SPEAKERMODE_MONO`, `FMOD_SPEAKERMODE_STEREO`.
• NGP - `FMOD_SPEAKERMODE_RAW`, `FMOD_SPEAKERMODE_MONO`, `FMOD_SPEAKERMODE_STEREO`.
• 3DS - `FMOD_SPEAKERMODE_RAW`, `FMOD_SPEAKERMODE_MONO`, `FMOD_SPEAKERMODE_STEREO`.
• Android - `FMOD_SPEAKERMODE_RAW`, `FMOD_SPEAKERMODE_MONO`, `FMOD_SPEAKERMODE_STEREO`.
• Wii - `FMOD_SPEAKERMODE_RAW`, `FMOD_SPEAKERMODE_MONO`, `FMOD_SPEAKERMODE_STEREO`.
• PS3 - `FMOD_SPEAKERMODE_7POINT1` only. The user cannot change the speaker mode on this platform.
• Xbox 360 - `FMOD_SPEAKERMODE_5POINT1` only. The user cannot change the speaker mode on this platform.

**NOTE!** Calling this function resets any speaker positions set with `System::set3DSpeakerPosition`, therefore this function must be called before calling `System::set3DSpeakerPosition`.
If `System::setSoftwareFormat` is called after this function with a valid output channel count, the speakermode is set to `FMOD_SPEAKERMODE_RAW`. If this function is called after `System::setSoftwareFormat`, then it will overwrite the channel count specified in that function.

The channel count that is overwritten for each speaker mode is as follows:
• `FMOD_SPEAKERMODE_RAW` - Channel count is unaffected.
• `FMOD_SPEAKERMODE_MONO` - Channel count is set to 1.
• `FMOD_SPEAKERMODE_STEREO` - Channel count is set to 2.
• `FMOD_SPEAKERMODE_QUAD` - Channel count is set to 4.
• `FMOD_SPEAKERMODE_SURROUND` - Channel count is set to 5.
• `FMOD_SPEAKERMODE_5POINT1` - Channel count is set to 6.
• `FMOD_SPEAKERMODE_7POINT1` - Channel count is set to 8.
• `FMOD_SPEAKERMODE_SRS5_1_MATRIX` - Channel count is set to 6.
• `FMOD_SPEAKERMODE_MYEARS` - Channel count is set to 8.

These channel counts are the channel width of the FMOD DSP system, and affect software mixed sounds (sounds created with `FMOD_SOFTWARE` flag) only.
Hardware sounds are not affected, but will still have the speaker mode...
appropriately set if possible. (On Windows the speaker mode is set by the user in the control panel, not by FMOD).

**NOTE! (Windows only)** Sound will not behave correctly unless your control panel has set the speaker mode to the correct setup. For example if `FMOD_SPEAKERMODE_7POINT1` is set on a speaker system that has been set to 'stereo' in the windows control panel, sounds can dissapear and come out of the wrong speaker. Make sure your users know about this.

If using WinMM output, note that some soundcard drivers do not support multichannel output correctly (i.e. Creative cards).

If using WASAPI the speaker mode will be forced to the control panel setting, you can call `System::getSpeakerMode` after `System::init` to verify.

Only DirectSound, WASAPI and ASIO have reliably working multichannel output.

If the speaker mode is not actually supported (even though the user set the speaker mode to 7.1 in Windows) the soundcard might not be able to handle it. You will get `FMOD_ERR_OUTPUT_CREATEBUFFER` error. Change the speaker mode to `FMOD_SPEAKERMODE_STEREO` and re-initialize if this happens.

To set the speaker mode to that of the windows control panel, use `System::getDriverCaps`. For example:

```cpp
FMOD_SPEAKERMODE speakermode;
FMOD_RESULT result;
result = system->getDriverCaps(0,0,0,0,&speakermode); // Get speaker mode
ERRCHECK(result);
result = system->setSpeakerMode(speakermode);
ERRCHECK(result);
```

This function cannot be called after FMOD is already activated with `System::init`.
It must be called before `System::init`, or after `System::close`.

**See Also**

- `System::getSpeakerMode`
- `FMOD_SPEAKERMODE`
- `System::init`
- `System::close`
- `System::setSoftwareFormat`
- `System::set3DSpeakerPosition`
- `System::getDriverCaps`
- `FMOD_RESULT`

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
**System::setStreamBufferSize**

Sets the internal buffersize for streams opened after this call. Larger values will consume more memory (see remarks), whereas smaller values may cause buffer under-run/starvation/stuttering caused by large delays in disk access (ie CDROM or netstream), or cpu usage in slow machines, or by trying to play too many streams at once.

**C++ Syntax**

```cpp
FMOD_RESULT System::setStreamBufferSize(
    unsigned int  filebuffersize,
    FMOD_TIMEUNIT filebuffersizetype
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_SetStreamBufferSize(
    FMOD_SYSTEM *  system,
    unsigned int  filebuffersize,
    FMOD_TIMEUNIT filebuffersizetype
);
```

**Parameters**

`filebuffersize`

Size of stream file buffer. Default is 16384 (`FMOD_TIMEUNIT_RAWBYTES`).

`filebuffersizetype`

Type of unit for stream file buffer size. Must be `FMOD_TIMEUNIT_MS`, `FMOD_TIMEUNIT_PCM`, `FMOD_TIMEUNIT_PCMBYTES` or `FMOD_TIMEUNIT_RAWBYTES`. Default is `FMOD_TIMEUNIT_RAWBYTES`.

**Return Values**
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

### Remarks

Note this function does not affect streams created with **FMOD_OPENUSER**, as the buffer size is specified in **System::createSound**. This function does not affect latency of playback. All streams are pre-buffered (unless opened with **FMOD_OPENONLY**), so they will always start immediately. Seek and Play operations can sometimes cause a reflush of this buffer.

If **FMOD_TIMEUNIT_RAWBYTES** is used, the memory allocated is 2 * the size passed in, because fmod allocates a double buffer. If **FMOD_TIMEUNIT_MS**, **FMOD_TIMEUNIT_PCM** or **FMOD_TIMEUNIT_PCMBYTES** is used, and the stream is infinite (such as a shoutcast netstream), or VBR, then FMOD cannot calculate an accurate compression ratio to work with when the file is opened. This means it will then base the buffersize on **FMOD_TIMEUNIT_PCMBYTES**, or in other words the number of PCM bytes, but this will be incorrect for some compressed formats. Use **FMOD_TIMEUNIT_RAWBYTES** for these type (infinite / undetermined length) of streams for more accurate read sizes.

Note to determine the actual memory usage of a stream, including sound buffer and other overhead, use **Memory_GetStats** before and after creating a sound.

Note that the stream may still stutter if the codec uses a large amount of cpu time, which impacts the smaller, internal 'decode' buffer. The decode buffer size is changeable via **FMOD_CREATESOUDINDEXINFO**.

### See Also

- **FMOD_TIMEUNIT**
- **System::createSound**
• System::getStreamBufferSize
• Sound::getOpenState
• Channel::setMute
• Memory_GetStats
• FMOD_CREATESAOUNDEXINFO

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
System::setUserData

Sets a user value that the System object will store internally. Can be retrieved with System::getUserData.

C++ Syntax

```cpp
FMOD_RESULT System::setUserData(  
    void *   userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetUserData(  
    FMOD_SYSTEM * system,  
    void *   userdata
);
```

Parameters

userdata

Address of user data that the user wishes stored within the System object.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using **System::getUserData** would help in the identification of the object.

**See Also**

- **System::getUserData**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
System::unloadPlugin

Unloads a plugin from memory.

C++ Syntax

```cpp
FMOD_RESULT System::unloadPlugin(
    unsigned int handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_UnloadPlugin(
    FMOD_SYSTEM * system,
    unsigned int handle
);
```

Parameters

handle

Handle to a pre-existing plugin.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::loadPlugin
Firelight Technologies FMOD Ex
System::unlockDSP

Mutual exclusion function to unlock the FMOD DSP engine (which runs asynchronously in another thread) and let it continue executing.

C++ Syntax

```cpp
FMOD_RESULT System::unlockDSP();
```

C Syntax

```c
FMOD_RESULT FMOD_System_UnlockDSP(FMOD_SYSTEM * system);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

The DSP engine must be locked with `System::lockDSP` before this function is called.

See Also

- `System::lockDSP`
Firelight Technologies FMOD Ex
System::update

Updates the FMOD system. This should be called once per 'game' tick, or once per frame in your application.

C++ Syntax

```cpp
FMOD_RESULT System::update();
```

C Syntax

```c
FMOD_RESULT FMOD_System_Update(FMOD_SYSTEM * system);
```

Parameters

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This updates the following things.

- 3D Sound. System::update must be called to get 3D positioning.
- Virtual voices. If more voices are played than there are real hardware/software voices, System::update must be called to handle the virtualization.
- *_NRT output modes. System::update must be called to drive the output for these output modes.
- FMOD_INIT_STREAM_FROM_UPDATE. System::update must be called to update the streamer if this flag has been used.
- Callbacks. System::update must be called to fire callbacks if they are specified.
• **FMOD_NONBLOCKING.** System::update must be called to make sounds opened with **FMOD_NONBLOCKING** flag to work properly.

• **FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED** callback. System::update must be called for this callback to trigger.

If **FMOD_OUTPUTTYPE_NOSOUND_NRT** or **FMOD_OUTPUTTYPE_WAVWRITER_NRT** output modes are used, this function also drives the software / DSP engine, instead of it running asynchronously in a thread as is the default behaviour. This can be used for faster than realtime updates to the decoding or DSP engine which might be useful if the output is the wav writer for example.

If **FMOD_INIT_STREAM_FROM_UPDATE** is used, this function will update the stream engine. Combining this with the non realtime output will mean smoother captured output.

**Warning!** Do not be tempted to call this function from a different thread to other FMOD commands! This is dangerous and will cause corruption/crashes. This function is not thread safe, and should be called from the same thread as the rest of the FMOD commands.

**See Also**

• **System::init**

• **FMOD_INITFLAGS**

• **FMOD_OUTPUTTYPE**

• **FMOD_MODE**
Firelight Technologies FMOD Ex
Sound Interface

Sound::addSyncPoint  Sound::deleteSyncPoint
Sound::get3DConeSettings
Sound::get3DCustomRolloff
Sound::get3DMinMaxDistance
Sound::getDefaults
Sound::getFormat
Sound::getLength
Sound::getLoopCount
Sound::getLoopPoints
Sound::getMemoryInfo
Sound::getMode
Sound::getMusicChannelVolume
Sound::getMusicNumChannels
Sound::getMusicSpeed
Sound::getName
Sound::getNumSubSounds
Sound::getNumSyncPoints
Sound::getNumTags
Sound::getOpenState
Sound::getSoundGroup
Sound::getSubSound
Sound::getSyncPoint
Sound::getSyncPointInfo
Sound::getSystemObject
Sound::getTag
Sound::getUserData
Sound::getVariations
Sound::lock
Sound::readData
Sound::release
Sound::seekData
Sound::set3DConeSettings
Sound::set3DCustomRolloff
Sound::set3DMinMaxDistance
Sound::setDefaults
Sound::setLoopCount
Sound::setLoopPoints
Sound::setMode
Sound::setMusicChannelVolume
Sound::setMusicSpeed
Sound::setSoundGroup
Sound::setSubSound
Sound::setSubSoundSentence
Sound::setUserData
Sound::setVariations
Sound::unlock
Firelight Technologies FMOD Ex
Sound::addSyncPoint

Adds a sync point at a specific time within the sound. These points can be user generated or can come from a wav file with embedded markers.

C++ Syntax

```cpp
FMOD_RESULT Sound::addSyncPoint(
    unsigned int offset,
    FMOD_TIMEUNIT offsettype,
    const char * name,
    FMOD_SYNCPOINT ** point
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_AddSyncPoint(
    FMOD_SOUND * sound,
    unsigned int offset,
    FMOD_TIMEUNIT offsettype,
    const char * name,
    FMOD_SYNCPOINT ** point
);
```

Parameters

- **offset**
- **offsettype**
- **name**
- **point**

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

In sound forge, a marker can be added a wave file by clicking on the timeline / ruler, and right clicking then selecting 'Insert Marker/Region'. Riff wrapped mp3 files are also supported.

See Also

- `Sound::getNumSyncPoints`
- `Sound::getSyncPoint`
- `Sound::getSyncPointInfo`
- `Sound::deleteSyncPoint`
Firelight Technologies FMOD Ex
Sound::deleteSyncPoint

Deletes a syncpoint within the sound. These points can be user generated or can come from a wav file with embedded markers.

C++ Syntax

```cpp
FMOD_RESULT Sound::deleteSyncPoint(
    FMOD_SYNCPOINT * point
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_DeleteSyncPoint(
    FMOD_SOUND * sound,
    FMOD_SYNCPOINT * point
);
```

Parameters

*point*

Address of an FMOD_SYNCPOINT object.

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

In sound forge, a marker can be added a wave file by clicking on the timeline / ruler, and right clicking then selecting 'Insert Marker/Region'. Riff wrapped mp3 files are also supported.
See Also

- `Sound::getNumSyncPoints`
- `Sound::getSyncPoint`
- `Sound::getSyncPointInfo`
- `Sound::addSyncPoint`
Firelight Technologies FMOD Ex
Sound::get3DConesettings

Retrieves the inside and outside angles of the sound projection cone.

C++ Syntax

```cpp
FMOD_RESULT Sound::get3DConesettings(
    float * insideconeangle,
    float * outsideconeangle,
    float * outsidevolume
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_Get3DConesettings(
    FMOD_SOUND * sound,
    float * insideconeangle,
    float * outsideconeangle,
    float * outsidevolume
);
```

Parameters

_insideconeangle_

Address of a variable that receives the inside angle of the sound projection cone, in degrees. This is the angle within which the sound is at its normal volume. Optional. Specify 0 or NULL to ignore.

_outsideconeangle_

Address of a variable that receives the outside angle of the sound projection cone, in degrees. This is the angle outside of which the sound is at its outside volume. Optional. Specify 0 or NULL to ignore.

_outsidevolume_

Address of a variable that receives the cone outside volume for this sound. Optional. Specify 0 or NULL to ignore.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [Sound::set3DConeSettings](#)
- [Channel::set3DConeSettings](#)
Firelight Technologies FMOD Ex
Sound::get3DCustomRolloff

Retrieves a pointer to the sound’s current custom rolloff curve.

C++ Syntax

```cpp
FMOD_RESULT Sound::get3DCustomRolloff(
    FMOD_VECTOR ** points,
    int * numpoints
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_Get3DCustomRolloff(
    FMOD_SOUND * sound,
    FMOD_VECTOR ** points,
    int * numpoints
);
```

Parameters

`points`

Address of a variable to receive the pointer to the current custom rolloff point list. Optional. Specify 0 or NULL to ignore.

`numpoints`

Address of a variable to receive the number of points in the current custom rolloff point list. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- **FMOD_VECTOR**
- **Sound::set3DCustomRolloff**
- **Channel::set3DCustomRolloff**
- **Channel::get3DCustomRolloff**
Firelight Technologies FMOD Ex
**Sound::get3DMinMaxDistance**

Retrieve the minimum and maximum audible distance for a sound.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::get3DMinMaxDistance(
    float * min,
    float * max
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_Get3DMinMaxDistance(
    FMOD_SOUND * sound,
    float * min,
    float * max
);
```

**Parameters**

*min*

Pointer to value to be filled with the minimum volume distance for the sound. See remarks for more on units. Optional. Specify 0 or NULL to ignore.

*max*

Pointer to value to be filled with the maximum volume distance for the sound. See remarks for more on units. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

A 'distance unit' is specified by System::set3DSettings. By default this is set to meters which is a distance scale of 1.0. See System::set3DSettings for more on this. The default units for minimum and maximum distances are 1.0 and 10,000.0f.

See Also

- Sound::set3DMinMaxDistance
- Channel::set3DMinMaxDistance
- Channel::get3DMinMaxDistance
- System::set3DSettings
Firelight Technologies FMOD Ex
Sound::getDefaults

Retrieves a sound's default attributes for when it is played on a channel with System::playSound.

C++ Syntax

```cpp
FMOD_RESULT Sound::getDefaults(
    float * frequency,
    float * volume,
    float * pan,
    int * priority
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetDefaults(
    FMOD_SOUND * sound,
    float * frequency,
    float * volume,
    float * pan,
    int * priority
);
```

Parameters

frequency

Address of a variable that receives the default frequency for the sound. Optional. Specify 0 or NULL to ignore.

volume

Address of a variable that receives the default volume for the sound. Result will be from 0.0 to 1.0. 0.0 = Silent, 1.0 = full volume. Default = 1.0. Optional. Specify 0 or NULL to ignore.

pan

Address of a variable that receives the default pan for the sound. Result will be
from -1.0 to +1.0. -1.0 = Full left, 0.0 = center, 1.0 = full right. Default = 0.0. Optional. Specify 0 or NULL to ignore.

**priority**

Address of a variable that receives the default priority for the sound when played on a channel. Result will be from 0 to 256. 0 = most important, 256 = least important. Default = 128. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [Sound::setDefaults](#)
- [System::createSound](#)
- [System::playSound](#)
Firelight Technologies FMOD Ex
Sound::getFormat

Returns format information about the sound.

C++ Syntax

```cpp
FMOD_RESULT Sound::getFormat(
    FMOD_SOUND_TYPE * type,
    FMOD_SOUND_FORMAT * format,
    int * channels,
    int * bits
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetFormat(
    FMOD_SOUND * sound,
    FMOD_SOUND_TYPE * type,
    FMOD_SOUND_FORMAT * format,
    int * channels,
    int * bits
);
```

Parameters

*type*

Address of a variable that receives the type of sound. Optional. Specify 0 or NULL to ignore.

*format*

Address of a variable that receives the format of the sound. Optional. Specify 0 or NULL to ignore.

*channels*

Address of a variable that receives the number of channels for the sound. Optional. Specify 0 or NULL to ignore.

*bits*
Address of a variable that receives the number of bits per sample for the sound. This corresponds to `FMOD_SOUND_FORMAT` but is provided as an integer format for convenience. Hardware compressed formats such as VAG, XADPCM, GCADPCM that stay compressed in memory will return 0. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- `FMOD_SOUND_TYPE`
- `FMOD_SOUND_FORMAT`
Firelight Technologies FMOD Ex
Sound::getLength

Retrieves the length of the sound using the specified time unit.

C++ Syntax

```cpp
FMOD_RESULT Sound::getLength(
    unsigned int * length,
    FMOD_TIMEUNIT lengthtype
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetLength(
    FMOD_SOUND * sound,
    unsigned int * length,
    FMOD_TIMEUNIT lengthtype
);
```

Parameters

*length*

Address of a variable that receives the length of the sound.

*lengthtype*

Time unit retrieve into the length parameter. See `FMOD_TIMEUNIT`.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks
Certain timeunits do not work depending on the file format. For example `FMOD_TIMEUNIT_MODORDER` will not work with an mp3 file. A length of 0xFFFFFFFF usually means it is of unlimited length, such as an internet radio stream or MOD/S3M/XM/IT file which may loop forever.

**Warning!** Using a VBR source that does not have an associated length information in milliseconds or pcm samples (such as MP3 or MOD/S3M/XM/IT) may return inaccurate lengths specify `FMOD_TIMEUNIT_MS` or `FMOD_TIMEUNIT_PCM`. If you want FMOD to retrieve an accurate length it will have to pre-scan the file first in this case. You will have to specify `FMOD_ACCURATETIME` when loading or opening the sound. This means there is a slight delay as FMOD scans the whole file when loading the sound to find the right length in milliseconds or pcm samples, and this also creates a seek table as it does this for seeking purposes.

**See Also**

- `FMOD_TIMEUNIT`
Firelight Technologies FMOD Ex
Sound::getLoopCount

Retrieves the current loop count value for the specified sound.

C++ Syntax

```cpp
FMOD_RESULT Sound::getLoopCount(
    int * loopcount
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetLoopCount(
    FMOD_SOUND * sound,
    int * loopcount
);
```

Parameters

`loopcount`

Address of a variable that receives the number of times a sound will loop by default before stopping. 0 = oneshot. 1 = loop once then stop. -1 = loop forever. Default = -1

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Unlike the channel loop count function, this function simply returns the value set with `Sound::setLoopCount`. It does not decrement as it plays (especially seeing
as one sound can be played multiple times).

See Also

- `Sound::setLoopCount`
Firelight Technologies FMOD Ex
Sound::getLoopPoints

Retrieves the loop points for a sound.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::getLoopPoints(
    unsigned int * loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int * loopend,
    FMOD_TIMEUNIT loopendtype
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_GetLoopPoints(
    FMOD_SOUND * sound,
    unsigned int * loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int * loopend,
    FMOD_TIMEUNIT loopendtype
);
```

**Parameters**

**loopstart**

Address of a variable to receive the loop start point. This point in time is played, so it is inclusive. Optional. Specify 0 or NULL to ignore.

**loopstarttype**

The time format used for the returned loop start point. See [FMOD_TIMEUNIT](#).

**loopend**

Address of a variable to receive the loop end point. This point in time is played, so it is inclusive. Optional. Specify 0 or NULL to ignore.

**loopendtype**
The time format used for the returned loop end point. See `FMOD_TIMEUNIT`.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- `FMOD_TIMEUNIT`
- `Sound::setLoopPoints`
Sound::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT Sound::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetMemoryInfo(
    FMOD_SOUND * sound,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

`memorybits`

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

`event_memorybits`

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

`memoryused`

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

`memoryused_details`
Optional. Specify 0 to ignore. Address of a user-allocated
`FMOD_MEMORY_USAGE_DETAILS` structure to be filled with detailed
memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the
`FMOD_RESULT` enumeration.

**Remarks**

See [System::getMemoryInfo](#) for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGE_DETAILS`
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
Sound::getMode

Retrieves the mode bits set by the codec and the user when opening the sound.

C++ Syntax

```cpp
FMOD_RESULT Sound::getMode(
    FMOD_MODE * mode
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetMode(
    FMOD_SOUND * sound,
    FMOD_MODE * mode
);
```

Parameters

`mode`

Address of a variable that receives the current mode for this sound.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Sound::setMode`
- `System::createSound`
- `Channel::setMode`
- `Channel::getMode`
Firelight Technologies FMOD Ex
Sound::getMusicChannelVolume

Retrieves the volume of a MOD/S3M/XM/IT/MIDI music channel volume.

C++ Syntax

```cpp
FMOD_RESULT Sound::getMusicChannelVolume(
    int channel,
    float * volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetMusicChannelVolume(
    FMOD_SOUND * sound,
    int channel,
    float * volume
);
```

Parameters

channel

MOD/S3M/XM/IT/MIDI music subchannel to retrieve the volume for.

volume

Address of a variable to receive the volume of the channel from 0.0 to 1.0. Default = 1.0.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Remarks

Use `Sound::getMusicNumChannels` to get the maximum number of music channels in the song.

See Also

- `Sound::getMusicNumChannels`
- `Sound::setMusicChannelVolume`
Firelight Technologies FMOD Ex
Sound::getMusicNumChannels

Gets the number of music channels inside a MOD/S3M/XM/IT/MIDI file.

C++ Syntax

```cpp
FMOD_RESULT Sound::getMusicNumChannels(
    int * numchannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetMusicNumChannels(
    FMOD_SOUND * sound,
    int * numchannels
);
```

Parameters

`numchannels`

Number of music channels used in the song.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Sound::setMusicChannelVolume`
- `Sound::getMusicChannelVolume`
Firelight Technologies FMOD Ex
Sound::getMusicSpeed

Sets the relative speed of MOD/S3M/XM/IT/MIDI music.

C++ Syntax

```cpp
FMOD_RESULT Sound::getMusicSpeed(
    float * speed
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetMusicSpeed(
    FMOD_SOUND * sound,
    float * speed
);
```

Parameters

speed

Address of a variable to receive the relative speed of the song from 0.01 to 100.0. 0.5 = half speed, 2.0 = double speed. Default = 1.0.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Setting a speed outside the bounds of 0.01 to 100.0 will not return an error, it will clamp the value.
See Also

- `Sound::setMusicSpeed`
- `Sound::setMusicChannelVolume`
- `Sound::getMusicChannelVolume`
Firelight Technologies FMOD Ex
Sound::getName

Retrieves the name of a sound.

C++ Syntax

```cpp
FMOD_RESULT Sound::getName(
    char * name,
    int namelen
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetName(
    FMOD_SOUND * sound,
    char * name,
    int namelen
);
```

Parameters

name

Address of a variable that receives the name of the sound.

namelen

Length in bytes of the target buffer to receive the string.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks
if `FMOD_LOWMEM` has been specified in `System::createSound`, this function will return "(null)".

**See Also**

- [System::createSound](#)
- [FMOD_MODE](#)
Firelight Technologies FMOD Ex
Sound::getNumSubSounds

Retrieves the number of subsounds stored within a sound.

C++ Syntax

```cpp
FMOD_RESULT Sound::getNumSubSounds(
    int * numsubsounds
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetNumSubSounds(
    FMOD_SOUND * sound,
    int * numsubsounds
);
```

Parameters

`numsubsounds`

Address of a variable that receives the number of subsounds stored within this sound.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

A format that has subsounds is usually a container format, such as FSB, DLS, MOD, S3M, XM, IT.
See Also

- Sound::getSubSound
Firelight Technologies FMOD Ex
Sound::getNumSyncPoints

Retrieves the number of sync points stored within a sound. These points can be user generated or can come from a wav file with embedded markers.

C++ Syntax

```cpp
FMOD_RESULT Sound::getNumSyncPoints(
    int * numsyncpoints
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetNumSyncPoints(
    FMOD_SOUND * sound,
    int * numsyncpoints
);
```

Parameters

`numsyncpoints`

Address of a variable to receive the number of sync points within this sound.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

In sound forge, a marker can be added a wave file by clicking on the timeline / ruler, and right clicking then selecting 'Insert Marker/Region'.
Riff wrapped mp3 files are also supported.
See Also

- `Sound::getSyncPoint`
- `Sound::getSyncPointInfo`
- `Sound::addSyncPoint`
- `Sound::deleteSyncPoint`
Firelight Technologies FMOD Ex
**Sound::getNumTags**

Retrieves the number of tags belonging to a sound.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::getNumTags(
    int * numtags,
    int * numtagsupdated
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_GetNumTags(
    FMOD_SOUND * sound,
    int * numtags,
    int * numtagsupdated
);
```

**Parameters**

*numtags*

Address of a variable that receives the number of tags in the sound. Optional. Specify 0 or NULL to ignore.

*numtagsupdated*

Address of a variable that receives the number of tags updated since this function was last called. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

The 'numtagsupdated' parameter can be used to check if any tags have been updated since last calling this function. This can be useful to update tag fields, for example from internet based streams, such as shoutcast or icecast where the name of the song might change.

See Also

- Sound::getTag
Firelight Technologies FMOD Ex
Sound::getOpenState

Retrieves the state a sound is in after FMOD_NONBLOCKING has been used to open it, or the state of the streaming buffer.

C++ Syntax

```cpp
FMOD_RESULT Sound::getOpenState(
    FMOD_OPENSTATE * openstate,
    unsigned int * percentbuffered,
    bool * starving,
    bool * diskbusy
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetOpenState(
    FMOD_SOUND * sound,
    FMOD_OPENSTATE * openstate,
    unsigned int * percentbuffered,
    FMOD_BOOL * starving,
    FMOD_BOOL * diskbusy
);
```

Parameters

openstate

Address of a variable that receives the open state of a sound. Optional. Specify 0 or NULL to ignore.

percentbuffered

Address of a variable that receives the percentage of the file buffer filled progress of a stream. Optional. Specify 0 or NULL to ignore.

starving

Address of a variable that receives the starving state of a sound. If a stream has decoded more than the stream file buffer has ready for it, it will return TRUE.
Optional. Specify 0 or NULL to ignore.

\textit{diskbusy}

Address of a variable that receives the disk busy state of a sound. That is, whether the disk is currently being accessed for the sound.

**Return Values**

If the function succeeds then the return value is \texttt{FMOD_OK}.

If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

\textbf{Note:} The return value will be the result of the asynchronous sound create. Use this to determine what happened if a sound failed to open.

\textbf{Note:} Always check 'openstate' to determine the state of the sound. Do not assume that if this function returns \texttt{FMOD\_OK} then the sound has finished loading.

**Remarks**

When a sound is opened with \texttt{FMOD\_NONBLOCKING}, it is opened and prepared in the background, or asynchronously. This allows the main application to execute without stalling on audio loads. This function will describe the state of the asynchronous load routine i.e. whether it has succeeded, failed or is still in progress.

If 'starving' is true, then you will most likely hear a stuttering/repeating sound as the decode buffer loops on itself and replays old data. Now that this variable exists, you can detect buffer underrun and use something like \texttt{Channel::setMute} to keep it quiet until it is not starving any more.

**See Also**

- \texttt{FMOD\_OPENSTATE}
• **FMOD_MODE**
• **Channel::setMute**
Firelight Technologies FMOD Ex
Sound::getSoundGroup

Retrieves the sound's current soundgroup.

C++ Syntax

```cpp
FMOD_RESULT Sound::getSoundGroup(
    FMOD::SoundGroup ** soundgroup
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetSoundGroup(
    FMOD_SOUND * sound,
    FMOD_SOUNDBANK ** soundgroup
);
```

Parameters

`soundgroup`

Address of a pointer to a SoundGroup to receive the sound's current soundgroup.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
By default a sound is located in the 'master sound group'. This can be retrieved with `System::getMasterSoundGroup`.

See Also

- `Sound::setSoundGroup`
• System::getMasterSoundGroup
Firelight Technologies FMOD Ex
**Sound::getSubSound**

Retrieves a handle to a Sound object that is contained within the parent sound.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::getSubSound(
    int index,
    FMOD::Sound ** subsound
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_GetSubSound(
    FMOD_SOUND * sound,
    int index,
    FMOD_SOUND ** subsound
);
```

**Parameters**

*index*

Index of the subsound to retrieve within this sound.

*subsound*

Address of a variable that receives the sound object specified.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**
If the sound is a stream and `FMOD_NONBLOCKING` was not used, then this call will perform a blocking seek/flush to the specified subsound.

If `FMOD_NONBLOCKING` was used to open this sound and the sound is a stream, FMOD will do a non blocking seek/flush and set the state of the subsound to `FMOD_OPENSTATE_SEEKING`.
The sound won't be ready to be used in this case until the state of the sound becomes `FMOD_OPENSTATE_READY` (or `FMOD_OPENSTATE_ERROR`).

See Also

- `Sound::getNumSubSounds`
- `Sound::setSubSound`
- `System::createSound`
- `FMOD_MODE`
- `FMOD_OPENSTATE`
Firelight Technologies FMOD Ex
Sound::getSyncPoint

Retrieve a handle to a sync point. These points can be user generated or can come from a wav file with embedded markers.

C++ Syntax

```cpp
FMOD_RESULT Sound::getSyncPoint(
    int      index,
    FMOD_SYNCPOINT ** point
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetSyncPoint(
    FMOD_SOUND * sound,
    int    index,
    FMOD_SYNCPOINT ** point
);
```

Parameters

`index`

Index of the sync point to retrieve. Use Sound::getNumSyncPoints to determine the number of syncpoints.

`point`

Address of a variable to receive a pointer to a sync point.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Remarks

In sound forge, a marker can be added a wave file by clicking on the timeline / ruler, and right clicking then selecting 'Insert Marker/Region'. Riff wrapped mp3 files are also supported.

See Also

- Sound::getNumSyncPoints
- Sound::getSyncPointInfo
- Sound::addSyncPoint
- Sound::deleteSyncPoint
Firelight Technologies FMOD Ex
Sound::getSyncPointInfo

Retrieves information on an embedded sync point. These points can be user generated or can come from a wav file with embedded markers.

C++ Syntax

```cpp
FMOD_RESULT Sound::getSyncPointInfo(
    FMOD_SYNCPOINT * point,
    char * name,
    int namelen,
    unsigned int * offset,
    FMOD_TIMEUNIT offsettype
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetSyncPointInfo(
    FMOD_SOUND * sound,
    FMOD_SYNCPOINT * point,
    char * name,
    int namelen,
    unsigned int * offset,
    FMOD_TIMEUNIT offsettype
);
```

Parameters

`point`

Pointer to a sync point. Use Sound::getSyncPoint to retrieve a syncpoint or Sound::addSyncPoint to create one.

`name`

Address of a variable to receive the name of the syncpoint. Optional. Specify 0 or NULL to ignore.

`namelen`

Size of buffer in bytes for name parameter. FMOD will only copy to this point if
the string is bigger than the buffer passed in. Specify 0 to ignore name parameter.

_offset_

Address of a variable to receive the offset of the syncpoint in a format determined by the offsettype parameter. Optional. Specify 0 or NULL to ignore.

_offsettype_

A timeunit parameter to determine a desired format for the offset parameter. For example the offset can be specified as pcm samples, or milliseconds.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

In sound forge, a marker can be added a wave file by clicking on the timeline / ruler, and right clicking then selecting 'Insert Marker/Region'. Riff wrapped mp3 files are also supported.

**See Also**

- **Sound::getNumSyncPoints**
- **Sound::getSyncPoint**
- **Sound::addSyncPoint**
- **Sound::deleteSyncPoint**
Firelight Technologies FMOD Ex
Sound::getSystemObject

Retrieves the parent System object that was used to create this object.

C++ Syntax

```cpp
FMOD_RESULT Sound::getSystemObject(
    FMOD::System ** system
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetSystemObject(
    FMOD_SOUND * sound,
    FMOD_SYSTEM ** system
);
```

Parameters

`system`

Address of a pointer that receives the System object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::createSound
Firelight Technologies FMOD Ex
Sound::getTag

Retrieves a descriptive tag stored by the sound, to describe things like the song name, author etc.

C++ Syntax

```cpp
FMOD_RESULT Sound::getTag(
    const char * name,
    int index,
    FMOD_TAG * tag
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetTag(
    FMOD_SOUND * sound,
    const char * name,
    int index,
    FMOD_TAG * tag
);
```

Parameters

name

Optional. Name of a tag to retrieve. Used to specify a particular tag if the user requires it. To get all types of tags leave this parameter as 0 or NULL.

index

Index into the tag list. If the name parameter is null, then the index is the index into all tags present, from 0 up to but not including the numtags value returned by Sound::getNumTags.

If name is not null, then index is the index from 0 up to the number of tags with the same name. For example if there were 2 tags with the name "TITLE" then you could use 0 and 1 to reference them.

Specifying an index of -1 returns new or updated tags. This can be used to pull tags out as they are added or updated.
tag

Pointer to a tag structure. This will receive

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

The number of tags available can be found with Sound::getNumTags. The way to display or retrieve tags can be done in 3 different ways.
All tags can be continuously retrieved by looping from 0 to the numtags value in Sound::getNumTags - 1. Updated tags will refresh automatically, and the 'updated' member of the FMOD_TAG structure will be set to true if a tag has been updated, due to something like a netstream changing the song name for example.
Tags could also be retrieved by specifying -1 as the index and only updating tags that are returned. If all tags are retrieved and this function is called the function will return an error of FMOD_ERR_TAGNOTFOUND.
Specific tags can be retrieved by specifying a name parameter. The index can be 0 based or -1 in the same fashion as described previously.

See Also

- Sound::getNumTags
- FMOD_TAG
Firelight Technologies FMOD Ex
Sound::getUserData

Retrieves the user value that was set by calling the Sound::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT Sound::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetUserData(
    FMOD_SOUND * sound,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the Sound::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- Sound::setUserData
Firelight Technologies FMOD Ex
Sound::getVariations

Retrieves the current playback behaviour variations of a sound.

C++ Syntax

```cpp
FMOD_RESULT Sound::getVariations(
    float * frequencyvar,
    float * volumevar,
    float * panvar
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_GetVariations(
    FMOD_SOUND * sound,
    float * frequencyvar,
    float * volumevar,
    float * panvar
);
```

Parameters

`frequencyvar`

Address of a variable to receive the frequency variation in hz. Frequency will play at its default frequency, plus or minus a random value within this range. Default = 0.0. Specify 0 or NULL to ignore.

`volumevar`

Address of a variable to receive the volume variation. 0.0 to 1.0. Sound will play at its default volume, plus or minus a random value within this range. Default = 0.0. Specify 0 or NULL to ignore.

`panvar`

Address of a variable to receive the pan variation. 0.0 to 2.0. Sound will play at its default pan, plus or minus a random value within this range. Pan is from -1.0 to +1.0 normally so the range can be a maximum of 2.0 in this case. Default = 0.
Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [Sound::setVariations](#)
Firelight Technologies FMOD Ex
**Sound::lock**

Returns a pointer to the beginning of the sample data for a sound.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::lock(  
    unsigned int offset,  
    unsigned int length,  
    void ** ptr1,  
    void ** ptr2,  
    unsigned int * len1,  
    unsigned int * len2
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_Lock(  
    FMOD_SOUND * sound,  
    unsigned int offset,  
    unsigned int length,  
    void ** ptr1,  
    void ** ptr2,  
    unsigned int * len1,  
    unsigned int * len2
);
```

**Parameters**

- **offset**

  Offset in *bytes* to the position you want to lock in the sample buffer.

- **length**

  Number of *bytes* you want to lock in the sample buffer.

- **ptr1**

  Address of a pointer that will point to the first part of the locked data.
ptr2
Address of a pointer that will point to the second part of the locked data. This will be null if the data locked hasn't wrapped at the end of the buffer.

len1
Length of data in bytes that was locked for ptr1

len2
Length of data in bytes that was locked for ptr2. This will be 0 if the data locked hasn't wrapped at the end of the buffer.

Return Values
If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks
You must always unlock the data again after you have finished with it, using Sound::unlock. With this function you get access to the RAW audio data, for example 8, 16, 24 or 32bit PCM data, mono or stereo data, and on consoles, vag, xadpcm or gcadpcm compressed data. You must take this into consideration when processing the data within the pointer.

See Also
- Sound::unlock
- System::createSound
Firelight Technologies FMOD Ex
**Sound::readData**

Reads data from an opened sound to a specified pointer, using the FMOD codec created internally. This can be used for decoding data offline in small pieces (or big pieces), rather than playing and capturing it, or loading the whole file at once and having to lock / unlock the data.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::readData(
    void * buffer,
    unsigned int lenbytes,
    unsigned int * read
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_ReadData(
    FMOD_SOUND * sound,
    void * buffer,
    unsigned int lenbytes,
    unsigned int * read
);
```

**Parameters**

* buffer

Address of a buffer that receives the decoded data from the sound.

* lenbytes

Number of bytes to read into the buffer.

* read

Number of bytes actually read.
Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

If too much data is read, it is possible **FMOD_ERR_FILE_EOF** will be returned, meaning it is out of data. The 'read' parameter will reflect this by returning a smaller number of bytes read than was requested.
As a sound already reads the whole file then closes it upon calling **System::createSound** (unless **System::createStream** or **FMOD_CREATESTREAM** is used), this function will not work because the file is no longer open.
Note that opening a stream makes it read a chunk of data and this will advance the read cursor. You need to either use **FMOD_OPENONLY** to stop the stream pre-buffering or call **Sound::seekData** to reset the read cursor.
If **FMOD_OPENONLY** flag is used when opening a sound, it will leave the file handle open, and FMOD will not read any data internally, so the read cursor will be at position 0. This will allow the user to read the data from the start.
As noted previously, if a sound is opened as a stream and this function is called to read some data, then you will 'miss the start' of the sound. **Channel::setPosition** will have the same result. These function will flush the stream buffer and read in a chunk of audio internally. This is why if you want to read from an absolute position you should use **Sound::seekData** and not the previously mentioned functions.
Remember if you are calling readData and seekData on a stream it is up to you to cope with the side effects that may occur. Information functions such as Channel::getPosition may give misleading results. Calling **Channel::setPosition** will reset and flush the stream, leading to the time values returning to their correct position.

NOTE! Thread safety. If you call this from another stream callback, or any other thread besides the main thread, make sure to put a critical section around the call, and another around **Sound::release** in case the sound is still being read from while releasing.
This function is thread safe to call from a stream callback or different thread as long as it doesn't conflict with a call to `Sound::release`.

**See Also**

- `Sound::seekData`
- `FMOD_MODE`
- `Channel::setPosition`
- `System::createSound`
- `System::createStream`
- `Sound::release`
Firelight Technologies FMOD Ex
Sound::release

Frees a sound object.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::release();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_Release(FMOD_SOUND * sound);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This will free the sound object and everything created under it.

If this is a stream that is playing as a subsound of another parent stream, then if this is the currently playing subsound (be it a normal subsound playback, or as part of a sentence), the whole stream will stop.

Note - This function will block if it was opened with `FMOD_NONBLOCKING` and hasn't finished opening yet.

**See Also**
- System::createSound
- Sound::getSubSound
Firelight Technologies FMOD Ex
Sound::seekData

Seeks a sound for use with data reading. This is not a function to 'seek a sound' for normal use. This is for use in conjunction with Sound::readData.

C++ Syntax

```cpp
FMOD_RESULT Sound::seekData(
    unsigned int  pcm
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_SeekData(
    FMOD_SOUND * sound,
    unsigned int  pcm
);
```

Parameters

`pcm`

Offset to seek to in PCM samples.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Note. If a stream is opened and this function is called to read some data, then it will advance the internal file pointer, so data will be skipped if you play the stream. Also calling position / time information functions will lead to misleading
results.
A stream can be reset before playing by setting the position of the channel (ie using `Channel::setPosition`), which will make it seek, reset and flush the stream buffer. This will make it sound correct again. Remember if you are calling readData and seekData on a stream it is up to you to cope with the side effects that may occur.

**See Also**

- `Sound::readData`
- `Channel::setPosition`
Firelight Technologies FMOD Ex
Sound::set3DConesSettings

Sets the inside and outside angles of the sound projection cone, as well as the volume of the sound outside the outside angle of the sound projection cone.

C++ Syntax

```cpp
FMOD_RESULT Sound::set3DConesSettings(
    float insideconeangle,
    float outsideconeangle,
    float outsidevolume
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_Set3DConesSettings(
    FMOD_SOUND * sound,
    float insideconeangle,
    float outsideconeangle,
    float outsidevolume
);
```

Parameters

**insideconeangle**

Inside cone angle, in degrees, from 0 to 360. This is the angle within which the sound is at its normal volume. Must not be greater than outsideconeangle. Default = 360.

**outsideconeangle**

Outside cone angle, in degrees, from 0 to 360. This is the angle outside of which the sound is at its outside volume. Must not be less than insideconeangle. Default = 360.

**outsidevolume**

Cone outside volume, from 0 to 1.0. Default = 1.0.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [Sound::get3DConeSettings](#)
- [Channel::set3DConeSettings](#)
Firelight Technologies FMOD Ex
**Sound::set3DCustomRolloff**

Point a sound to use a custom rolloff curve. Must be used in conjunction with `FMOD_3D_CUSTOMROLLOFF` flag to be activated.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::set3DCustomRolloff(
    FMOD_VECTOR * points,
    int numpoints
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_Set3DCustomRolloff(
    FMOD_SOUND * sound,
    FMOD_VECTOR * points,
    int numpoints
);
```

**Parameters**

*points*

An array of `FMOD_VECTOR` structures where x = distance and y = volume from 0.0 to 1.0. z should be set to 0.

*numpoints*

The number of points in the array.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Note! This function does not duplicate the memory for the points internally. The pointer you pass to FMOD must remain valid until there is no more use for it. Do not free the memory while in use, or use a local variable that goes out of scope while in use.

Points must be sorted by distance! Passing an unsorted list to FMOD will result in an error.

Set the points parameter to 0 or NULL to disable the points. If `FMOD_3D_CUSTOMROLLOFF` is set and the rolloff curve is 0, FMOD will revert to inverse curve rolloff.

Min and maxdistance are meaningless when `FMOD_3D_CUSTOMROLLOFF` is used and the values are ignored.

Here is an example of a custom array of points.

```cpp
FMOD_VECTOR curve[3] =
{
    { 0.0f, 1.0f, 0.0f },
    { 2.0f, 0.2f, 0.0f },
    { 20.0f, 0.0f, 0.0f }
};
```

x represents the distance, y represents the volume. z is always 0.
Distances between points are linearly interpolated.
Note that after the highest distance specified, the volume in the last entry is used from that distance onwards.

See Also

- `FMOD_MODE`
- `FMOD_VECTOR`
- `Sound::get3DCustomRolloff`
- `Channel::set3DCustomRolloff`
- `Channel::get3DCustomRolloff`
Firelight Technologies FMOD Ex
Sound::set3DMinMaxDistance

Sets the minimum and maximum audible distance for a sound.

MinDistance is the minimum distance that the sound emitter will cease to continue growing louder at (as it approaches the listener).
Within the mindistance it stays at the constant loudest volume possible. Outside of this mindistance it begins to attenuate.
MaxDistance is the distance a sound stops attenuating at. Beyond this point it will stay at the volume it would be at maxdistance units from the listener and will not attenuate any more.
MinDistance is useful to give the impression that the sound is loud or soft in 3d space. An example of this is a small quiet object, such as a bumblebee, which you could set a mindistance of to 0.1 for example, which would cause it to attenuate quickly and disappear when only a few meters away from the listener.
Another example is a jumbo jet, which you could set to a mindistance of 100.0, which would keep the sound volume at max until the listener was 100 meters away, then it would be hundreds of meters more before it would fade out.

In summary, increase the mindistance of a sound to make it 'louder' in a 3d world, and decrease it to make it 'quieter' in a 3d world.
Maxdistance is effectively obsolete unless you need the sound to stop fading out at a certain point. Do not adjust this from the default if you dont need to.
Some people have the confusion that maxdistance is the point the sound will fade out to, this is not the case.

C++ Syntax

```cpp
FMOD_RESULT Sound::set3DMinMaxDistance(
    float min,
    float max
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_Set3DMinMaxDistance(
    FMOD_SOUND * sound,
```
Parameters

min

The sound's minimum volume distance in "units". See remarks for more on units.

max

The sound's maximum volume distance in "units". See remarks for more on units.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

A 'distance unit' is specified by **System::set3DSettings**. By default this is set to meters which is a distance scale of 1.0.
See **System::set3DSettings** for more on this.
The default units for minimum and maximum distances are 1.0 and 10,000.0f.

See Also

- **Sound::get3DMinMaxDistance**
- **Channel::set3DMinMaxDistance**
- **Channel::get3DMinMaxDistance**
- **System::set3DSettings**

```c
float min,
float max
);
```
Firelight Technologies FMOD Ex
**Sound::setDefaults**

Sets a sound's default attributes, so when it is played it uses these values without having to specify them later for each channel each time the sound is played.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::setDefaults(
    float frequency,
    float volume,
    float pan,
    int priority
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_SetDefaults(
    FMOD_SOUND * sound,
    float frequency,
    float volume,
    float pan,
    int priority
);
```

**Parameters**

*frequency*

Default playback frequency for the sound, in hz. (ie 44100hz).

*volume*

Default volume for the sound. 0.0 to 1.0. 0.0 = Silent, 1.0 = full volume. Default = 1.0.

*pan*

Default pan for the sound. -1.0 to +1.0. -1.0 = Full left, 0.0 = center, 1.0 = full right. Default = 0.0.
priority

Default priority for the sound when played on a channel. 0 to 256. 0 = most important, 256 = least important. Default = 128.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

There are no 'ignore' values for these parameters. Use Sound::getDefaults if you want to change only 1 and leave others unaltered.

See Also

- `Sound::getDefaults`
- `System::playSound`
- `System::createSound`
Firelight Technologies FMOD Ex
Sound::setLoopCount

Sets a sound, by default, to loop a specified number of times before stopping if its mode is set to FMOD_LOOP_NORMAL or FMOD_LOOP_BIDI.

C++ Syntax

```cpp
FMOD_RESULT Sound::setLoopCount(
    int  loopcount
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_SetLoopCount(
    FMOD_SOUND * sound,
    int  loopcount
);
```

Parameters

`loopcount`

Number of times to loop before stopping. 0 = oneshot. 1 = loop once then stop. -1 = loop forever. Default = -1

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function does not affect FMOD_HARDWARE based sounds that are not streamable.
FMODSOFTWARE based sounds or any type of sound created with System::CreateStream or FMOD_CREATESTREAM will support this function.

**Issues with streamed audio.** (Sounds created with System::createStream or FMOD_CREATESTREAM). When changing the loop count, sounds created with System::createStream or FMOD_CREATESTREAM may already have been pre-buffered and executed their loop logic ahead of time, before this call was even made. This is dependant on the size of the sound versus the size of the stream decode buffer. See FMOD_CREATESOUNDEXINFO. If this happens, you may need to refill the stream buffer. To do this, you can call Channel::setPosition which forces a refill of the stream buffer. Note this will usually only happen if you have sounds or looppoints that are smaller than the stream decode buffer size. Otherwise you will not normally encounter any problems.

**See Also**

- `Sound::getLoopCount`
- `System::setStreamBufferize`
- `FMOD_CREATESOUNDEXINFO`
Firelight Technologies FMOD Ex
**Sound::setLoopPoints**

Sets the loop points within a sound.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::setLoopPoints(
    unsigned int loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int loopend,
    FMOD_TIMEUNIT loopendtype
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_SetLoopPoints(
    FMOD_SOUND *sound,
    unsigned int loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int loopend,
    FMOD_TIMEUNIT loopendtype
);
```

**Parameters**

*loopstart*

The loop start point. This point in time is played, so it is inclusive.

*loopstarttype*

The time format used for the loop start point. See **FMOD_TIMEUNIT**.

*loopend*

The loop end point. This point in time is played, so it is inclusive.

*loopendtype*

The time format used for the loop end point. See **FMOD_TIMEUNIT**.
**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Not supported by static sounds created with `FMOD_HARDWARE`. Supported by sounds created with `FMOD_SOFTWARE`, or sounds of any type (hardware or software) created with `System::createStream` or `FMOD_CREATESTREAM`.

If a sound was 44100 samples long and you wanted to loop the whole sound, loopstart would be 0, and loopend would be 44099, not 44100. You wouldn't use milliseconds in this case because they are not sample accurate.

If loop end is smaller or equal to loop start, it will result in an error.
If loop start or loop end is larger than the length of the sound, it will result in an error.

**Issues with streamed audio.** (Sounds created with with `System::createStream` or `FMOD_CREATESTREAM`)

When changing the loop points, sounds created with `System::createStream` or `FMOD_CREATESTREAM` may already have been pre-buffered and executed their loop logic ahead of time, before this call was even made. This is dependant on the size of the sound versus the size of the stream *decode* buffer. See `FMOD_CREATESOUNDEXINFO`.

If this happens, you may need to refill the stream buffer. To do this, you can call `Channel::setPosition` which forces a refill of the stream buffer.

Note this will usually only happen if you have sounds or looppoinnts that are smaller than the stream *decode* buffer size. Otherwise you will not normally encounter any problems.

**See Also**
- FMOD_TIMEUNIT
- FMOD_MODE
- Sound::getLoopPoints
- Sound::setLoopCount
- System::createStream
- System::setStreamBufferSize
- Channel::setPosition
- FMOD_CREATESOUNDEXINFO
Firelight Technologies FMOD Ex
**Sound::setMode**

Sets or alters the mode of a sound.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::setMode(
    FMOD_MODE mode
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_SetMode(
    FMOD_SOUND * sound,
    FMOD_MODE mode
);
```

**Parameters**

*mode*

Mode bits to set.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

When calling this function, note that it will only take effect when the sound is played again with `System::playSound`. Consider this mode the 'default mode' for when the sound plays, not a mode that will suddenly change all currently playing instances of this sound.
Flags supported:
FMOD_LOOP_OFF
FMOD_LOOP_NORMAL
FMOD_LOOP_BIDI (only works with sounds created with
FMODSOFTWARE. Otherwise it will behave as FMOD_LOOP_NORMAL)
FMOD_3D_HEADRELATIVE
FMOD_3D_WORLDRELATIVE
FMOD_2D (see notes for win32 hardware voices)
FMOD_3D (see notes for win32 hardware voices)
FMOD_3D_INVERSEROLLOFF
FMOD_3D_LINEARROLLOFF
FMOD_3D_LINEARSQUAREROLLOFF
FMOD_3D_CUSTOMROLLOFF
FMOD_3D_IGNOREGEOMETRY
FMOD_DONTRESTOREVIRTUAL

Issues with streamed audio. (Sounds created with with System::createStream or
FMOD_CREATESTREAM). When changing the loop mode, sounds created
with System::createStream or FMOD_CREATESTREAM may already have
been pre-buffered and executed their loop logic ahead of time, before this call
was even made.
This is dependant on the size of the sound versus the size of the stream decode
buffer. See FMOD_CREATESOUNDEXINFO.
If this happens, you may need to flush the stream buffer. To do this, you can
call Channel::setPosition which forces a flush of the stream buffer.
Note this will usually only happen if you have sounds or looppoints that are
smaller than the stream decode buffer size. Otherwise you will not normally
encounter any problems.

If FMOD_3D_IGNOREGEOMETRY is not specified, the flag will be cleared if
it was specified previously.

See Also

- FMOD_MODE
- Sound::getMode
- System::setStreamBufferSize
- `System::playSound`
- `System::createStream`
- `Channel::setPosition`
- `FMOD_CREATESOUNDEXINFO`
Firelight Technologies FMOD Ex
**Sound::setMusicChannelVolume**

Sets the volume of a MOD/S3M/XM/IT/MIDI music channel volume.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::setMusicChannelVolume(
    int channel,
    float volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_SetMusicChannelVolume(
    FMOD_SOUND * sound,
    int channel,
    float volume
);
```

**Parameters**

- **channel**
  
  MOD/S3M/XM/IT/MIDI music subchannel to set a linear volume for.

- **volume**
  
  Volume of the channel from 0.0 to 1.0. Default = 1.0.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**
Use `Sound::getMusicNumChannels` to get the maximum number of music channels in the song.

**See Also**

- `Sound::getMusicNumChannels`
- `Sound::getMusicChannelVolume`
Firelight Technologies FMOD Ex
Sound::setMusicSpeed

Sets the relative speed of MOD/S3M/XM/IT/MIDI music.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::setMusicSpeed(
    float speed
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_SetMusicSpeed(
    FMOD_SOUND * sound,
    float speed
);
```

**Parameters**

`speed`

Relative speed of the song from 0.01 to 100.0. 0.5 = half speed, 2.0 = double speed. Default = 1.0.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Setting a speed outside the bounds of 0.01 to 100.0 will not return an error, it will clamp the value.
See Also

- Sound::getMusicSpeed
- Sound::setMusicChannelVolume
- Sound::getMusicChannelVolume
Firelight Technologies FMOD Ex
**Sound::setSoundGroup**

Moves the sound from its existing SoundGroup to the specified sound group.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::setSoundGroup(
    FMOD::SoundGroup * soundgroup
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_SetSoundGroup(
    FMOD_SOUND * sound,
    FMOD_SOUNDGROUP * soundgroup
);
```

**Parameters**

* soundgroup

Address of a SoundGroup object to move the sound to.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

By default a sound is located in the 'master sound group'. This can be retrieved with **System::getMasterSoundGroup**.
Putting a sound in a sound group (or just using the master sound group) allows for functionality like limiting a group of sounds to a certain number of playbacks.
(see `SoundGroup::setMaxAudible`).

See Also

- `Sound::getSoundGroup`
- `System::getMasterSoundGroup`
- `System::createSoundGroup`
- `SoundGroup::setMaxAudible`
Sound::setSubSound

Assigns a sound as a 'subsound' of another sound. A sound can contain other sounds. The sound object that is issuing the command will be the 'parent' sound.

C++ Syntax

```cpp
FMOD_RESULT Sound::setSubSound(
    int index,
    FMOD::Sound * subsound
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_SetSubSound(
    FMOD_SOUND * sound,
    int index,
    FMOD_SOUND * subsound
);
```

Parameters

`index`

Index within the sound to set the new sound to as a 'subsound'.

`subsound`

Sound object to set as a subsound within this sound.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

If a subsound is set which is a member of a sentence (See `Sound::setSubSoundSentence`), the loop points of the sentence will be reset to a loopstart of 0 and a loopend of the length of the sentence minus 1. This means any sentence must have its loop points re-set by the user if this function is called in this case.

See Also

- `Sound::getNumSubSounds`
- `Sound::getSubSound`
- `Sound::setSubSoundSentence`
Firelight Technologies FMOD Ex
Sound::setSubSoundSentence

For any sound that has subsounds, this function will determine the order of playback of these subsounds, and it will play / stitch together the subsounds without gaps. This is a very useful feature for those users wanting to do seamless / gapless stream playback. (ie sports commentary, gapless playback media players etc).

C++ Syntax

```cpp
FMOD_RESULT Sound::setSubSoundSentence(
    int * subsoundlist,
    int numsubsounds
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_SetSubSoundSentence(
    FMOD_SOUND * sound,
    int * subsoundlist,
    int numsubsounds
);
```

Parameters

`subsoundlist`

Pointer to an array of indicies which are the subsounds to play. One subsound can be included in this list multiple times if required.

`numsubsounds`

Number of indicies inside the subsoundlist array.

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the
Remarks

The currently playing subsound in a sentence can be found with `Channel::getPosition` and the timeunit `FMOD_TIMEUNIT_SENTENCE_SUBSOUND`. This is useful for displaying the currently playing track of a cd in a whole CD sentence for example. For realtime stitching purposes, it is better to know the buffered ahead of time subsound index. This can be done by adding the flag (using bitwise OR) `FMOD_TIMEUNIT_BUFFERED`. If `FMOD_ERR_SUBSOUND_MODE` is returned, then `FMOD_CREATECOMPRESSESAMPLE` has been used on a child sound, but not on the parent sound, or vice versa. Another cause of this is trying to mix a static sample with a streaming sound. Mode bits to do with this must match. A user can change subsounds that are not playing at the time, to do dynamic stitching/sentencing of sounds.

See Also

- `System::playSound`
- `Sound::getSubSound`
- `Channel::getPosition`
- `FMOD_TIMEUNIT`
Firelight Technologies FMOD Ex
Sound::setUserData

Sets a user value that the Sound object will store internally. Can be retrieved with Sound::getUserData.

C++ Syntax

```cpp
FMOD_RESULT Sound::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Sound_SetUserData(
    FMOD_SOUND * sound,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the Sound object.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using `Sound::getUserData` would help in the identification of the object.

**See Also**

- `Sound::getUserData`
Firelight Technologies FMOD Ex
**Sound::setVariations**

Changes the playback behaviour of a sound by allowing random variations to playback parameters to be set.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::setVariations(
    float frequencyvar,
    float volumevar,
    float panvar
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_SetVariations(
    FMOD_SOUND * sound,
    float frequencyvar,
    float volumevar,
    float panvar
);
```

**Parameters**

`frequencyvar`

Frequency variation in hz. Frequency will play at its default frequency, plus or minus a random value within this range. Default = 0.0.

`volumevar`

Volume variation. 0.0 to 1.0. Sound will play at its default volume, plus or minus a random value within this range. Default = 0.0.

`panvar`

Pan variation. 0.0 to 2.0. Sound will play at its default pan, plus or minus a random value within this range. Pan is from -1.0 to +1.0 normally so the range can be a maximum of 2.0 in this case. Default = 0.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [Sound::getVariations](#)
Firelight Technologies FMOD Ex
**Sound::unlock**

Releases previous sample data lock from `Sound::lock`.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::unlock(
    void * ptr1,
    void * ptr2,
    unsigned int len1,
    unsigned int len2
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_Unlock(
    FMOD_SOUND * sound,
    void * ptr1,
    void * ptr2,
    unsigned int len1,
    unsigned int len2
);
```

**Parameters**

`ptr1`

Pointer to the 1st locked portion of sample data, from `Sound::lock`.

`ptr2`

Pointer to the 2nd locked portion of sample data, from `Sound::lock`.

`len1`

Length of data in bytes that was locked for `ptr1`

`len2`

Length of data in bytes that was locked for `ptr2`. This will be 0 if the data locked hasn't wrapped at the end of the buffer.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Call this function after data has been read/written to from `Sound::lock`. This function will do any post processing necessary and if needed, send it to sound ram.

See Also

- `Sound::lock`
- `System::createSound`
Firelight Technologies FMOD Ex
Channel Interface

Channel::addDSP
Channel::get3DAttributes
Channel::get3DConeOrientation
Channel::get3DConeSettings
Channel::get3DCustomRolloff
Channel::get3DDistanceFilter
Channel::get3DDopplerLevel
Channel::get3DMinMaxDistance
Channel::get3DOcclusion
Channel::get3DPanLevel
Channel::get3DSpread
Channel::getAudibility
Channel::getChannelGroup
Channel::getCurrentSound
Channel::getDSPHead
Channel::getDelay
Channel::getFrequency
Channel::getIndex
Channel::getInputChannelMix
Channel::getLoopCount
Channel::getLoopPoints
Channel::getLowPassGain
Channel::getMemoryInfo
Channel::getMode
Channel::getMute
Channel::getPan
Channel::getPaused
Channel::getPosition
Channel::getPriority
Channel::getReverbProperties
Channel::getSpeakerLevels
Channel::getSpeakerMix
Channel::getSpectrum
Channel::getSystemObject
Channel::getUserData
Channel::getVolume
Channel::getWaveData
Channel::isPlaying
Channel::isVirtual
Channel::set3DAttributes
Channel::set3DConoOrientation
Channel::set3DConeSettings
Channel::set3DCustomRolloff
Channel::set3DDistanceFilter
Channel::set3DDopplerLevel
Channel::set3DMinMaxDistance
Channel::set3DOcclusion
Channel::set3DPanLevel
Channel::set3DSpread
Channel::setCallback
Channel::setChannelGroup
Channel::setDelay
Channel::setFrequency
Channel::setInputChannelMix
Channel::setLoopCount
Channel::setLoopPoints
Channel::setLowPassGain
Channel::setMode
Channel::setMute
Channel::setPan
Channel::setPaused
Channel::setPosition
Channel::setPriority
Channel::setReverbProperties
Channel::setSpeakerLevels
Channel::setSpeakerMix
Channel::setUserData
Channel::setVolume
Channel::stop
Firelight Technologies FMOD Ex
Channel::addDSP

This function adds a pre-created DSP unit or effect to the head of the Channel DSP chain.

C++ Syntax

```cpp
FMOD_RESULT Channel::addDSP(
    FMOD::DSP * dsp,
    FMOD::DSPConnection ** connection
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_AddDSP(
    FMOD_CHANNEL * channel,
    FMOD_DSP * dsp,
    FMOD_DSPCONNECTION ** connection
);
```

Parameters

dsp

A pointer to a pre-created DSP unit to be inserted at the head of the Channel DSP chain.

connection

A pointer to the connection involved between the Channel DSP head and the specified dsp unit. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
**Remarks**

This function is a wrapper function to insert a DSP unit at the top of the Channel DSP chain. It disconnects the head unit from its input, then inserts the unit at the head and reconnects the previously disconnected input back as as an input to the new unit.

**Note:** The connection pointer retrieved here will become invalid if you disconnect the 2 dsp units that use it.

**See Also**

- Channel::getDSPHead
- System::createDSP
- System::createDSPByType
- System::createDSPByPlugin
- System::addDSP
- ChannelGroup::addDSP
- DSP::remove
Firelight Technologies FMOD Ex
Channel::get3DAttributes

Retrieves the position and velocity of a 3d channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::get3DAttributes(
    FMOD_VECTOR * pos,
    FMOD_VECTOR * vel
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Get3DAttributes(
    FMOD_CHANNEL * channel,
    FMOD_VECTOR * pos,
    FMOD_VECTOR * vel
);
```

Parameters

`pos`

Address of a variable that receives the position in 3D space of the channel. Optional. Specify 0 or NULL to ignore.

`vel`

Address of a variable that receives the velocity in 'distance units per second' in 3D space of the channel. See remarks. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

A 'distance unit' is specified by System::set3DSettings. By default this is set to meters which is a distance scale of 1.0.

See Also

- Channel::set3DAttributes
- FMODVECTOR
- System::set3DSettings
Firelight Technologies FMOD Ex
Channel::get3DConeOrientation

Retrieves the orientation of the sound projection cone for this channel.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::get3DConeOrientation(
    FMOD_VECTOR * orientation
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_Get3DConeOrientation(
    FMOD_CHANNEL * channel,
    FMOD_VECTOR * orientation
);
```

**Parameters**

orientation

Address of a variable that receives the orientation of the sound projection cone. The vector information represents the center of the sound cone.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- [Channel::set3DConeOrientation](#)
Firelight Technologies FMOD Ex
Channel::get3DConeSettings

Retrieves the inside and outside angles of the sound projection cone.

C++ Syntax

FMOD_RESULT Channel::get3DConeSettings(
    float * insideconeangle,
    float * outsideconeangle,
    float * outsidevolume
);

C Syntax

FMOD_RESULT FMOD_Channel_Get3DConeSettings(
    FMOD_CHANNEL * channel,
    float * insideconeangle,
    float * outsideconeangle,
    float * outsidevolume
);

Parameters

insideconeangle

Address of a variable that receives the inside angle of the sound projection cone, in degrees. This is the angle within which the sound is at its normal volume. Optional. Specify 0 or NULL to ignore.

outsideconeangle

Address of a variable that receives the outside angle of the sound projection cone, in degrees. This is the angle outside of which the sound is at its outside volume. Optional. Specify 0 or NULL to ignore.

outsidevolume

Address of a variable that receives the cone outside volume for this channel. Optional. Specify 0 or NULL to ignore.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- Channel::set3DConeSettings
- Sound::get3DConeSettings
Firelight Technologies FMOD Ex
Channel::get3DCustomRolloff

Retrieves a pointer to the sound’s current custom rolloff curve.

C++ Syntax

```cpp
FMOD_RESULT Channel::get3DCustomRolloff(
    FMOD_VECTOR ** points,
    int * numpoints
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Get3DCustomRolloff(
    FMOD_CHANNEL * channel,
    FMOD_VECTOR ** points,
    int * numpoints
);
```

Parameters

points

Address of a variable to receive the pointer to the current custom rolloff point list. Optional. Specify 0 or NULL to ignore.

numpoints

Address of a variable to receive the number of points in the current custom rolloff point list. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
See Also

- FMOD_VECTOR
- Channel::set3DCustomRolloff
- Sound::set3DCustomRolloff
- Sound::get3DCustomRolloff
Firelight Technologies FMOD Ex
Channel::get3DDistanceFilter

C++ Syntax

```cpp
FMOD_RESULT Channel::get3DDistanceFilter(
    bool * custom,
    float * customLevel,
    float * centerFreq
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Get3DDistanceFilter(
    FMOD_CHANNEL * channel,
    FMOD_BOOL * custom,
    float * customLevel,
    float * centerFreq
);
```

Parameters

custom

customLevel

centerFreq

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Firelight Technologies FMOD Ex
Channel::get3DDopplerLevel

Retrieves the current 3D doppler level for the channel set by Channel::set3DDopplerLevel.

C++ Syntax

```cpp
FMOD_RESULT Channel::get3DDopplerLevel(
    float * level
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Get3DDopplerLevel(
    FMOD_CHANNEL * channel,
    float * level
);
```

Parameters

`level`

Address of a variable to receives the current doppler scale for this channel. 0 = No doppler. 1 = Normal doppler. 5 = max. Default = 1.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Channel::set3DDopplerLevel
Firelight Technologies FMOD Ex
Channel::get3DMinMaxDistance

Retrieves the current minimum and maximum audible distance for a channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::get3DMinMaxDistance(
    float * mindistance,
    float * maxdistance
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Get3DMinMaxDistance(
    FMOD_CHANNEL * channel,
    float * mindistance,
    float * maxdistance
);
```

Parameters

`mindistance`

Pointer to a floating point value to store mindistance. Optional. Specify 0 or NULL to ignore.

`maxdistance`

Pointer to a floating point value to store maxdistance. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- Channel::set3DMinMaxDistance
- System::set3DSettings
- Sound::set3DMinMaxDistance
Firelight Technologies FMOD Ex
Channel::get3DOcclusion

Retrieves the occlusion factors for a channel.

C++ Syntax

```
FMOD_RESULT Channel::get3DOcclusion(
    float * directocclusion,
    float * reverbocclusion
);
```

C Syntax

```
FMOD_RESULT FMOD_Channel_Get3DOcclusion(
    FMOD_CHANNEL * channel,
    float * directocclusion,
    float * reverbocclusion
);
```

Parameters

`directocclusion`

Address of a variable that receives the occlusion factor for a voice for the direct path. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0. Optional. Specify 0 or NULL to ignore.

`reverbocclusion`

Address of a variable that receives the occlusion factor for a voice for the reverb mix. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- Channel::set3DOcclusion
- ChannelGroup::get3DOcclusion
Firelight Technologies FMOD Ex
Channel::get3DPanLevel

Retrieves the current 3D mix level for the channel set by Channel::set3DPanLevel.

C++ Syntax

`FMOD_RESULT Channel::get3DPanLevel( float * level );`

C Syntax

`FMOD_RESULT FMOD_Channel_Get3DPanLevel( FMOD_CHANNEL * channel, float * level );`

Parameters

`level`

0 = Sound pans according to Channel::setSpeakerMix. 1 = Sound pans according to 3d position. Default = 1 (all by 3d position).

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- Channel::set3DPanLevel
- Channel::setSpeakerMix
Firelight Technologies FMOD Ex
Channel::get3DSpread

Retrieves the stereo (and above) spread angle specified by 
Channel::set3DSpread.

C++ Syntax

```cpp
FMOD_RESULT Channel::get3DSpread(
    float * angle
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Get3DSpread(
    FMOD_CHANNEL * channel,
    float * angle
);
```

Parameters

angle

Address of a variable that receives the spread angle for subchannels. 0 = all subchannels are located at the same position. 360 = all subchannels are located at the opposite position.

Return Values

If the function succeeds then the return value is FMOD_OK. 
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

• Channel::set3DSpread
Firelight Technologies FMOD Ex
Channel::getAudibility

Returns the combined volume of the channel after 3d sound, volume, channel group volume and geometry occlusion calculations have been performed on it.

C++ Syntax

```cpp
FMOD_RESULT Channel::getAudibility(
    float * audibility
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetAudibility(
    FMOD_CHANNEL * channel,
    float * audibility
);
```

Parameters

`audibility`

Address of a variable that receives the channel audibility value.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This does not represent the waveform, just the calculated volume based on 3d distance, occlusion, volume and channel group volume. This value is used by the FMOD Ex virtual channel system to order its channels between real and virtual.
See Also

- `Channel::setVolume`
- `Channel::getVolume`
- `ChannelGroup::setVolume`
- `ChannelGroup::getVolume`
- `Channel::set3DOcclusion`
- `Channel::get3DOcclusion`
- `Channel::set3DAttributes`
- `Channel::get3DAttributes`
Firelight Technologies FMOD Ex
Channel::getChannelGroup

Retrieves the currently assigned channel group for the channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getChannelGroup(
    FMOD::ChannelGroup ** channelgroup
);```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetChannelGroup(
    FMOD_CHANNEL * channel,
    FMOD_CHANNELGROUP ** channelgroup
);```

Parameters

`channelgroup`

Address of a variable to receive a pointer to the currently assigned channel group.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Channel::setChannelGroup
Firelight Technologies FMOD Ex
Channel::getCurrentSound

Returns the currently playing sound for this channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getCurrentSound(
    FMOD::Sound ** sound
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetCurrentSound(
    FMOD_CHANNEL * channel,
    FMOD_SOUND ** sound
);
```

Parameters

`sound`

Address of a variable that receives the pointer to the currently playing sound for this channel.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

If a sound is not playing the returned pointer will be 0 or NULL.
See Also

- System::playSound
- System::playDSP
Firelight Technologies FMOD Ex
Channel::getDSPHead

Returns a pointer to the DSP unit head node that handles software mixing for this channel.
Only applicable to channels playing sounds created with FMOD_SOFTWARE.

C++ Syntax

```cpp
FMOD_RESULT Channel::getDSPHead(
    FMOD::DSP ** dsp
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetDSPHead(
    FMOD_CHANNEL * channel,
    FMOD_DSP ** dsp
);
```

Parameters

dsp

Address of a variable that receives pointer to the current head DSP unit for this channel.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

By default a channel DSP unit usually contains 1 input, which is the wavetable
input. If \texttt{System::playDSP} has been used then the input to the channel head unit will be the unit that was specified in the call. See the tutorials for more information on DSP networks and how to manipulate them.

\textbf{See Also}

- \texttt{System::createDSP}
- \texttt{System::createDSPByType}
- \texttt{System::playDSP}
Firelight Technologies FMOD Ex
Channel::getDelay

Gets the currently set delay values.

C++ Syntax

```cpp
FMOD_RESULT Channel::getDelay(
    FMOD_DELAYTYPE delaytype,
    unsigned int * delayhi,
    unsigned int * delaylo
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetDelay(
    FMOD_CHANNEL * channel,
    FMOD_DELAYTYPE delaytype,
    unsigned int * delayhi,
    unsigned int * delaylo
);
```

Parameters

*delaytype*

See [FMOD_DELAYTYPE](#). This determines what delayhi and delaylo will represent.

*delayhi*

Address of a variable to receive the top (most significant) 32 bits of a 64bit number representing the time.

*delaylo*

Address of a variable to receive the bottom (least significant) 32 bits of a 64bit number representing the time.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

**Note!** Only works with sounds created with **FMOD_SOFTWARE**.

If **FMOD_DELAYTYPE_DSPCLOCK_START** is used, this will be the value of the DSP clock time at the time **System::playSound** was called, if the user has not called **Channel::setDelay**.

What is the 'dsp clock'? The DSP clock represents the output stream to the soundcard, and is incremented by the output rate every second (though of course with much finer granularity than this). So if your output rate is 48khz, the DSP clock will increment by 48000 per second.

The hi and lo values represent this 64bit number, with the delaylo representing the least significant 32bits and the delayhi value representing the most significant 32bits.

See Also

- **FMOD_DELAYTYPE**
- **Channel::setDelay**
- **System::playSound**
Firelight Technologies FMOD Ex
Channel::getFrequency

Returns the frequency in HZ of the channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getFrequency(
    float * frequency
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetFrequency(
    FMOD_CHANNEL * channel,
    float * frequency
);
```

Parameters

`frequency`

Address of a variable that receives the current frequency of the channel in HZ.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Channel::setFrequency
Firelight Technologies FMOD Ex
Channel::getIndex

Retrieves the internal channel index for a channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getIndex(
    int * index
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetIndex(
    FMOD_CHANNEL * channel,
    int * index
);
```

Parameters

`index`

Address of a variable to receive the channel index. This will be from 0 to the value specified in `System::init` minus 1.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Note that working with channel indicies directly is not recommended. It is recommended that you use `FMOD_CHANNEL_FREE` for the index in `System::playSound` to use FMOD's channel manager.
See Also

- System::playSound
- System::init
Firelight Technologies FMOD Ex
Channel::getInputChannelMix

Retrieves the incoming levels for a channel in a sound.
A mono sound has 1 input channel, a stereo has 2, etc. It depends on what type of sound is playing on the channel at the time.

C++ Syntax

```cpp
FMOD_RESULT Channel::getInputChannelMix(
    float * levels,
    int numlevels
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetInputChannelMix(
    FMOD_CHANNEL * channel,
    float * levels,
    int numlevels
);
```

Parameters

**levels**

Address of an array of float volume levels, from 0.0 to 1.0. These represent the incoming channels for the sound playing on the channel at the time.

**numlevels**

Number of floats to receive into the array. Maximum = the maximum number of input channels specified in `System::setSoftwareFormat`.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

This does not affect which speakers the sound is routed to. This can be used in conjunction with functions like Channel::setPan, Channel::setSpeakerMix, Channel::setSpeakerLevels.
This function only scales the input channels from the sound.

See Also

- Channel::setInputChannelMix
- Channel::setPan
- Channel::setSpeakerMix
- Channel::setSpeakerLevels
- System::setSoftwareFormat
Firelight Technologies FMOD Ex
Channel::getLoopCount

Retrieves the current loop count for the specified channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getLoopCount(
    int * loopcount
);
```

C Syntax

```
FMOD_RESULT FMOD_Channel_GetLoopCount(
    FMOD_CHANNEL * channel,
    int * loopcount
);
```

Parameters

`loopcount`

Address of a variable that receives the number of times a channel will loop before stopping. 0 = oneshot. 1 = loop once then stop. -1 = loop forever. Default = -1

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function retrieves the current loop countdown value for the channel being played.
This means it will decrement until reaching 0, as it plays. To reset the value, use `Channel::setLoopCount`.

See Also

- `Channel::setLoopCount`
Firelight Technologies FMOD Ex
Channel::getLoopPoints

Retrieves the loop points for a channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getLoopPoints(
    unsigned int * loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int * loopend,
    FMOD_TIMEUNIT loopendtype
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetLoopPoints(
    FMOD_CHANNEL * channel,
    unsigned int * loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int * loopend,
    FMOD_TIMEUNIT loopendtype
);
```

Parameters

*loopstart*

Address of a variable to receive the loop start point. This point in time is played, so it is inclusive. Optional. Specify 0 or NULL to ignore.

*loopstarttype*

The time format used for the returned loop start point. See `FMOD_TIMEUNIT`.

*loopend*

Address of a variable to receive the loop end point. This point in time is played, so it is inclusive. Optional. Specify 0 or NULL to ignore.

*loopendtype*
The time format used for the returned loop end point. See `FMOD_TIMEUNIT`.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- `FMOD_TIMEUNIT`
- `Channel::setLoopPoints`
Firelight Technologies FMOD Ex
Channel::getLowPassGain

Retrieves the gain of the dry signal when lowpass filtering is applied.

C++ Syntax

```cpp
FMOD_RESULT Channel::getLowPassGain(
    float * gain
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetLowPassGain(
    FMOD_CHANNEL * channel,
    float * gain
);
```

Parameters

*gain*

Address of a variable to receive the gain level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Channel::setLowPassGain`
Firelight Technologies FMOD Ex
Channel::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```c++
FMOD_RESULT Channel::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetMemoryInfo(
    FMOD_CHANNEL * channel,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

event_memorybits

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

memoryused

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated \texttt{FMOD\_MEMORY\_USAGE\_DETAILS} structure to be filled with detailed memory usage information about this object.

\textbf{Return Values}

If the function succeeds then the return value is \texttt{FMOD\_OK}. If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

\textbf{Remarks}

See \texttt{System::getMemoryInfo} for more details.

\textbf{See Also}

- \texttt{FMOD\_MEMBITS}
- \texttt{FMOD\_EVENT\_MEMBITS}
- \texttt{FMOD\_MEMORY\_USAGE\_DETAILS}
- \texttt{System::getMemoryInfo}
Firelight Technologies FMOD Ex
**Channel::getMode**

Retrieves the current mode bit flags for the current channel.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getMode(
    FMOD_MODE * mode
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetMode(
    FMOD_CHANNEL * channel,
    FMOD_MODE * mode
);
```

**Parameters**

`mode`

Address of a an `FMOD_MODE` variable that receives the current mode for this channel.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- `Channel::setMode`
Firelight Technologies FMOD Ex
Channel::getMute

Returns the current mute status of the channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getMute(
    bool * mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetMute(
    FMOD_CHANNEL * channel,
    FMOD_BOOL * mute
);
```

Parameters

*mute*

true = channel is muted (silent), false = channel is at normal volume.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Channel::setMute
Firelight Technologies FMOD Ex
Channel::getPan

Returns the pan position of the channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getPan(
    float * pan
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetPan(
    FMOD_CHANNEL * channel,
    float * pan
);
```

Parameters

`pan`

Address of a variable to receive the left/right pan level for the channel, from -1.0 to 1.0 inclusive. -1.0 = Full left, 1.0 = full right. Default = 0.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Channel::setPan
Firelight Technologies FMOD Ex
Channel::getPaused

Retrieves the paused state of the channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getPaused(
    bool * paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetPaused(
    FMOD_CHANNEL * channel,
    FMOD_BOOL * paused
);
```

Parameters

`paused`

Address of a variable that receives the current paused state. true = the sound is paused. false = the sound is not paused.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Channel::setPaused`
Firelight Technologies FMOD Ex
Channel::getPosition

Returns the current PCM offset or playback position for the specified channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getPosition(
    unsigned int * position,
    FMOD_TIMEUNIT postype
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGetPosition(
    FMOD_CHANNEL * channel,
    unsigned int * position,
    FMOD_TIMEUNIT postype
);
```

Parameters

`position`

Address of a variable that receives the position of the sound.

`postype`

Time unit to retrieve into the position parameter. See `FMOD_TIMEUNIT`.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks
Certain timeunits do not work depending on the file format. For example \texttt{FMOD\_TIMEUNIT\_MODORDER} will not work with an mp3 file. A PCM sample is a unit of measurement in audio that contains the data for one audible element of sound. 1 sample might be 16bit stereo, so 1 sample contains 4 bytes. 44,100 samples of a 44khz sound would represent 1 second of data. If the sound is a stream, and if you are using \texttt{FMOD\_NONBLOCKING}, note that you have used \texttt{Channel::setPosition}, then the stream will go into \texttt{FMOD\_OPENSTATE\_SETPOSITION} state (see \texttt{Sound::getOpenState}) and sound commands will return \texttt{FMOD\_ERR\_NOTREADY}. \texttt{Channel::getPosition} will not update until this non-blocking setPosition operation has completed.

\textbf{See Also}

- \texttt{Channel::setPosition}
- \texttt{FMOD\_TIMEUNIT}
- \texttt{Sound::getLength}
Firelight Technologies FMOD Ex
Channel::getPriority

Retrieves the current priority for this channel.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getPriority(
    int * priority
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetPriority(
    FMOD_CHANNEL * channel,
    int * priority
);
```

**Parameters**

`priority`

Address of a variable that receives the current channel priority. 0 to 256 inclusive. 0 = most important. 256 = least important. Default = 128.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- [Channel::setPriority](#)
Firelight Technologies FMOD Ex
Channel::getReverbProperties

Retrieves the current reverb properties for this channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getReverbProperties(
    FMOD_REVERB_CHANNELPROPERTIES * prop
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetReverbProperties(
    FMOD_CHANNEL * channel,
    FMOD_REVERB_CHANNELPROPERTIES * prop
);
```

Parameters

`prop`

Address of a variable to receive the `FMOD_REVERB_CHANNELPROPERTIES` information. Remember to clear this structure before calling this function, or specify the correct reverb instance with `FMOD_REVERB_CHANNELFLAGS`.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Note! It is important to clear this structure to 0 before calling, or at least specify
which reverb instance you are talking about by using the Flags member of 
**FMOD_REVERB_CHANNELPROPERTIES**, otherwise you may get an 
**FMOD_ERR_REVERB_INSTANCE** error.

**See Also**

- [Channel::setReverbProperties](#)
- **FMOD_REVERB_CHANNELPROPERTIES**
- **FMOD_REVERB_CHANNELFLAGS**
Firelight Technologies FMOD Ex
Channel::getSpeakerLevels

Retrieves the current pan matrix level settings from Channel::setSpeakerLevels.

C++ Syntax

```cpp
FMOD_RESULT Channel::getSpeakerLevels(
    FMOD_SPEAKER speaker,
    float * levels,
    int numlevels
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetSpeakerLevels(
    FMOD_CHANNEL * channel,
    FMOD_SPEAKER speaker,
    float * levels,
    int numlevels
);
```

Parameters

`speaker`

The speaker id to get the levels for. This can be cast to an integer if you are using a device with more than the pre-defined speaker range.

`levels`

Address of a variable that receives the current levels for the channel. This is an array of floating point values. The destination array size can be specified with the numlevels parameter.

`numlevels`

Number of floats in the destination array.
**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This function does not return level values reflecting `Channel::setVolume`. Volume is a separate scalar to the pan matrix levels.

**See Also**

- `Channel::setSpeakerLevels`
- `Channel::setPan`
- `Channel::setVolume`
Firelight Technologies FMOD Ex
Channel::getSpeakerMix

Sets the channel's speaker volume levels for each speaker individually.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getSpeakerMix(
    float * frontleft,
    float * frontright,
    float * center,
    float * lfe,
    float * backleft,
    float * backright,
    float * sideleft,
    float * sideright
);```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetSpeakerMix(
    FMOD_CHANNEL * channel,
    float * frontleft,
    float * frontright,
    float * center,
    float * lfe,
    float * backleft,
    float * backright,
    float * sideleft,
    float * sideright
);```

**Parameters**

*frontleft*

Address of a variable to receive the current volume level for this channel in the front left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume, up to 5.0 = 5x amplification.

*ftrontright*

Address of a variable to receive the current volume level for this channel in the
front right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume, up to 5.0 = 5x amplification.

center

Address of a variable to receive the current volume level for this channel in the center speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume, up to 5.0 = 5x amplification.

lfe

Address of a variable to receive the current volume level for this channel in the subwoofer speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume, up to 5.0 = 5x amplification.

backleft

Address of a variable to receive the current volume level for this channel in the back left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume, up to 5.0 = 5x amplification.

backright

Address of a variable to receive the current volume level for this channel in the back right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume, up to 5.0 = 5x amplification.

sideleft

Address of a variable to receive the current volume level for this channel in the side left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume, up to 5.0 = 5x amplification.

sideright

Address of a variable to receive the current volume level for this channel in the side right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume, up to 5.0 = 5x amplification.
**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

For 3D sound, the values set here are not representative of the 3d mix. For 3D sound this function is mainly for retrieving the LFE value if it was set by the user. This function is not affected by **Channel::setSpeakerLevels**. This function only returns the levels set by **Channel::setSpeakerMix**.

**See Also**

- [Channel::setSpeakerMix](#)
- [Channel::setSpeakerLevels](#)
Firelight Technologies FMOD Ex
Channel::getSpectrum

Retrieves the spectrum from the currently playing output signal for the current channel only.

C++ Syntax

```cpp
FMOD_RESULT Channel::getSpectrum(
    float * spectrumarray,
    int numvalues,
    int channeloffset,
    FMOD_DSP_FFT_WINDOW windowtype
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetSpectrum(
    FMOD_CHANNEL * channel,
    float * spectrumarray,
    int numvalues,
    int channeloffset,
    FMOD_DSP_FFT_WINDOW windowtype
);
```

Parameters

**spectrumarray**

Address of a variable that receives the spectrum data. This is an array of floating point values. Data will range is 0.0 to 1.0. Decibels = 10.0f * (float)log10(val) * 2.0f; See remarks for what the data represents.

**numvalues**

Size of array in floating point values being passed to the function. Must be a power of 2. (ie 128/256/512 etc). Min = 64. Max = 8192.

**channeloffset**

Channel of the signal to analyze. If the signal is multichannel (such as a stereo
output), then this value represents which channel to analyze. On a stereo signal 0 = left, 1 = right.

**windowtype**

"Pre-FFT" window method. This filters the PCM data before entering the spectrum analyzer to reduce transient frequency error for more accurate results. See **FMOD_DSP_FFT_WINDOW** for different types of fft window techniques possible and for a more detailed explanation.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

The larger the numvalues, the more CPU the FFT will take. Choose the right value to trade off between accuracy / speed.

The larger the numvalues, the more 'lag' the spectrum will seem to inherit. This is because the FFT window size stretches the analysis back in time to what was already played. For example if the numvalues size happened to be 44100 and the output rate was 44100 it would be analyzing the past second of data, and giving you the average spectrum over that time period.

If you are not displaying the result in dB, then the data may seem smaller than it should be. To display it you may want to normalize the data - that is, find the maximum value in the resulting spectrum, and scale all values in the array by 1 / max. (ie if the max was 0.5f, then it would become 1).

To get the spectrum for both channels of a stereo signal, call this function twice, once with channeloffset = 0, and again with channeloffset = 1. Then add the spectrums together and divide by 2 to get the average spectrum for both channels.

**What the data represents.**

To work out what each entry in the array represents, use this formula
entry_hz = (output_rate / 2) / numvalues

The array represents amplitudes of each frequency band from 0hz to the nyquist rate. The nyquist rate is equal to the output rate divided by 2.
For example when FMOD is set to 44100hz output, the range of represented frequencies will be 0hz to 22049hz, a total of 22050hz represented.
If in the same example, 1024 was passed to this function as the numvalues, each entry's contribution would be as follows.

entry_hz = (44100 / 2) / 1024
entry_hz = 21.53 hz

Note: This function only displays data for sounds playing that were created with FMODSOFTWARE. FMOD_HARDWARE based sounds are played using the sound card driver and are not accessible.

See Also

- FMOD_DSP_FFT_WINDOW
- System::getSpectrum
- ChannelGroup::getSpectrum
- System::getWaveData
Firelight Technologies FMOD Ex
Channel::getSystemObject

Retrieves the parent System object that was used to create this object.

C++ Syntax

```cpp
FMOD_RESULT Channel::getSystemObject(
    FMOD::System ** system
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetSystemObject(
    FMOD_CHANNEL * channel,
    FMOD_SYSTEM ** system
);
```

Parameters

**system**

Address of a variable that receives the System object.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- **System::playSound**
Firelight Technologies FMOD Ex
Channel::getUserData

Retrieves the user value that that was set by calling the Channel::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT Channel::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetUserData(
    FMOD_CHANNEL * channel,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the Channel::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

NOTE: If this channel was spawned by the event system then its user data field will be set, by the event system, to the event instance handle that spawned it. Use
this function to go from an arbitrary channel back up to the event that owns it.

See Also

- Channel::setUserData
Firelight Technologies FMOD Ex
Channel::getVolume

Retrieves the volume level for the channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getVolume(
    float * volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetVolume(
    FMOD_CHANNEL * channel,
    float * volume
);
```

Parameters

volume

Address of a variable to receive the channel volume level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- Channel::setVolume
Firelight Technologies FMOD Ex
Channel::getWaveData

Retrieves a pointer to a block of PCM data that represents the currently playing waveform on this channel. This function is useful for a very easy way to plot an oscilloscope.

C++ Syntax

```cpp
FMOD_RESULT Channel::getWaveData(
    float * wavearray,
    int numvalues,
    int channeloffset
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetWaveData(
    FMOD_CHANNEL * channel,
    float * wavearray,
    int numvalues,
    int channeloffset
);
```

Parameters

wavearray

Address of a variable that receives the currently playing waveform data. This is an array of floating point values.

numvalues

Number of floats to write to the array. Maximum value = 16384.

channeloffset

Offset into multichannel data. Mono channels use 0. Stereo channels use 0 = left, 1 = right. More than stereo use the appropriate index.
Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

This is the actual resampled pcm data window at the time the function is called.

Do not use this function to try and display the whole waveform of the sound, as this is more of a 'snapshot' of the current waveform at the time it is called, and could return the same data if it is called very quickly in succession.
See the DSP API to capture a continual stream of wave data as it plays, or see **Sound::lock** / **Sound::unlock** if you want to simply display the waveform of a sound.

This function allows retrieval of left and right data for a stereo sound individually. To combine them into one signal, simply add the entries of each seperate buffer together and then divide them by 2.
**Note:** This function only displays data for sounds playing that were created with **FMOD_SOFTWARE, FMOD_HARDWARE** based sounds are played using the sound card driver and are not accessable.

See Also

- **Channel::getSpectrum**
- **ChannelGroup::getWaveData**
- **System::getWaveData**
- **Sound::lock**
- **Sound::unlock**
Firelight Technologies FMOD Ex
Channel::isPlaying

Returns the playing state for the current channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::isPlaying(
    bool * isplaying
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_IsPlaying(
    FMOD_CHANNEL * channel,
    FMOD_BOOL * isplaying
);
```

Parameters

`isplaying`

Address of a variable that receives the current channel's playing status. true = the channel is currently playing a sound. false = the channel is not playing a sound.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [System::playSound](#)
- [System::playDSP](#)
Firelight Technologies FMOD Ex
Channel::isVirtual

Returns the current channel's status of whether it is virtual (emulated) or not due to FMOD Ex's virtual channel management system.

C++ Syntax

```cpp
FMOD_RESULT Channel::isVirtual(
    bool * isvirtual
);```

C Syntax

```c
FMOD_RESULT FMOD_Channel_IsVirtual(
    FMOD_CHANNEL * channel,
    FMOD_BOOL * isvirtual
);```

Parameters

`isvirtual`

Address of a variable that receives the current channel's virtual status. `true =` the channel is inaudible and currently being emulated at no cpu cost. `false =` the channel is a real hardware or software voice and should be audible.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Virtual channels are not audible, because there are no more real hardware or
software channels available.
If you are plotting virtual voices vs real voices graphically, and wondering why
FMOD sometimes chooses seemingly random channels to be virtual that are
usually far away, that is because they are probably silent. It doesn't matter which
are virtual and which are not if they are silent. Virtual voices are not calculation
on 'closest to listener' calculation, they are based on audibility. See the tutorial in
the FMOD Ex documentation for more information on virtual channels.

See Also

- System::playSound
Firelight Technologies FMOD Ex
Channel::set3DAttributes

Sets the position and velocity of a 3d channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DAttributes(
    const FMOD_VECTOR * pos,
    const FMOD_VECTOR * vel
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DAttributes(
    FMOD_CHANNEL * channel,
    const FMOD_VECTOR * pos,
    const FMOD_VECTOR * vel
);
```

Parameters

`pos`

Position in 3D space of the channel. Specifying 0 / null will ignore this parameter.

`vel`

Velocity in 'distance units per second' in 3D space of the channel. See remarks. Specifying 0 / null will ignore this parameter.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

A 'distance unit' is specified by `System::set3DSettings`. By default this is set to meters which is a distance scale of 1.0.

For a stereo 3d sound, you can set the spread of the left/right parts in speaker space by using `Channel::set3DSpread`.

See Also

- `Channel::get3DAttributes`
- `FMOD_VECTOR`
- `System::set3DSettings`
- `Channel::set3DSpread`
Firelight Technologies FMOD Ex
Channel::set3DConeOrientation

Sets the orientation of the sound projection cone.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DConeOrientation(
    FMOD_VECTOR * orientation
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DConeOrientation(
    FMOD_CHANNEL * channel,
    FMOD_VECTOR * orientation
);
```

Parameters

orientation

Pointer to an `FMOD_VECTOR` defining the coordinates of the sound cone orientation vector.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function has no effect unless the cone angle and cone outside volume have also been set to values other than the default.
See Also

- Channel::get3DConeOrientation
- Channel::set3DConeSettings
- Sound::set3DConeSettings
- FMOD_VECTOR
Firelight Technologies FMOD Ex
Channel::set3DConeSettings

Sets the inside and outside angles of the sound projection cone, as well as the volume of the sound outside the outside angle of the sound projection cone.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DConeSettings(
    float insideconeangle,
    float outsideconeangle,
    float outsidevolume
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DConeSettings(
    FMOD_CHANNEL * channel,
    float insideconeangle,
    float outsideconeangle,
    float outsidevolume
);
```

Parameters

*insideconeangle*

Inside cone angle, in degrees. This is the angle within which the sound is at its normal volume. Must not be greater than outsideconeangle. Default = 360.

*outsideconeangle*

Outside cone angle, in degrees. This is the angle outside of which the sound is at its outside volume. Must not be less than insideconeangle. Default = 360.

*outsidevolume*

Cone outside volume, from 0 to 1.0. Default = 1.0.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [Channel::get3DConeSettings](#)
- [Channel::set3DConeOrientation](#)
- [Sound::set3DConeSettings](#)
Firelight Technologies FMOD Ex
Channel::set3DCustomRolloff

Point a channel to use a custom rolloff curve. Must be used in conjunction with
FMOD_3D_CUSTOMROLLOFF flag to be activated.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DCustomRolloff(
    FMOD_VECTOR * points,
    int numpoints
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DCustomRolloff(
    FMOD_CHANNEL * channel,
    FMOD_VECTOR * points,
    int numpoints
);
```

Parameters

points

An array of FMOD_VECTOR structures where x = distance and y = volume from 0.0 to 1.0. z should be set to 0.

numpoints

The number of points in the array.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the
FMODResultado enumeration.
Remarks

Note! This function does not duplicate the memory for the points internally. The pointer you pass to FMOD must remain valid until there is no more use for it. Do not free the memory while in use, or use a local variable that goes out of scope while in use.

Points must be sorted by distance! Passing an unsorted list to FMOD will result in an error.

Set the points parameter to 0 or NULL to disable the points. If FMOD_3D_CUSTOMROLLOFF is set and the rolloff curve is 0, FMOD will revert to inverse curve rolloff.

Min and maxdistance are meaningless when FMOD_3D_CUSTOMROLLOFF is used and the values are ignored.

Here is an example of a custom array of points.

```c
FMOD_VECTOR curve[3] =
{
    { 0.0f, 1.0f, 0.0f },
    { 2.0f, 0.2f, 0.0f },
    { 20.0f, 0.0f, 0.0f }
};
```

x represents the distance, y represents the volume. z is always 0. Distances between points are linearly interpolated. Note that after the highest distance specified, the volume in the last entry is used from that distance onwards.

See Also

- FMOD_MODE
- FMOD_VECTOR
- Channel::get3DCustomRolloff
- Sound::set3DCustomRolloff
- Sound::get3DCustomRolloff
Firelight Technologies FMOD Ex
Channel::set3DDistanceFilter

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DDistanceFilter(
    bool custom,
    float customLevel,
    float centerFreq
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DDistanceFilter(
    FMOD_CHANNEL * channel,
    FMOD_BOOL custom,
    float customLevel,
    float centerFreq
);
```

Parameters

- `custom`
- `customLevel`
- `centerFreq`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Firelight Technologies FMOD Ex
Channel::set3DDopplerLevel

Sets the channel specific doppler scale for the channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DDopplerLevel(
    float level
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DDopplerLevel(
    FMOD_CHANNEL * channel,
    float level
);
```

Parameters

`level`

0 = No doppler. 1 = Normal doppler. 5 = max. Default = 1.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Channel::get3DDopplerLevel`
Firelight Technologies FMOD Ex
Channel::set3DMinMaxDistance

Sets the minimum and maximum audible distance for a channel.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::set3DMinMaxDistance(
    float mindistance,
    float maxdistance
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_Set3DMinMaxDistance(
    FMOD_CHANNEL * channel,
    float mindistance,
    float maxdistance
);
```

**Parameters**

*mindistance*

The channel's minimum volume distance in "units". See remarks for more on units.

*maxdistance*

The channel's maximum volume distance in "units". See remarks for more on units.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
**Remarks**

MinDistance is the minimum distance that the sound emitter will cease to continue growing louder at (as it approaches the listener). Within the mindistance it stays at the constant loudest volume possible. Outside of this mindistance it begins to attenuate. MaxDistance is the distance a sound stops attenuating at. Beyond this point it will stay at the volume it would be at maxdistance units from the listener and will not attenuate any more.

MinDistance is useful to give the impression that the sound is loud or soft in 3d space. An example of this is a small quiet object, such as a bumblebee, which you could set a mindistance of to 0.1 for example, which would cause it to attenuate quickly and dissappear when only a few meters away from the listener. Another example is a jumbo jet, which you could set to a mindistance of 100.0, which would keep the sound volume at max until the listener was 100 meters away, then it would be hundreds of meters more before it would fade out.

In summary, increase the mindistance of a sound to make it 'louder' in a 3d world, and decrease it to make it 'quieter' in a 3d world. maxdistance is effectively obsolete unless you need the sound to stop fading out at a certain point. Do not adjust this from the default if you dont need to.

Some people have the confusion that maxdistance is the point the sound will fade out to, this is not the case.

A 'distance unit' is specified by `System::set3DSettings`. By default this is set to meters which is a distance scale of 1.0. The default units for minimum and maximum distances are 1.0 and 10000.0f. Volume drops off at mindistance / distance.

To define the min and max distance per sound and not per channel use `Sound::set3DMinMaxDistance`.

If `FMOD_3D_CUSTOMROLLOFF` is used, then these values are stored, but ignored in 3d processing.
See Also

- Channel::get3DMinMaxDistance
- System::set3DSettings
- Sound::set3DMinMaxDistance
Firelight Technologies FMOD Ex
Channel::set3DOcclusion

Sets the occlusion factors for a channel. If the FMOD geometry engine is not being used, this function can be called to produce the same audible effects, just without the built in polygon processing. FMOD's internal geometry engine calls this function.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DOcclusion(
    float directocclusion,
    float reverbocclusion
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DOcclusion(
    FMOD_CHANNEL * channel,
    float directocclusion,
    float reverbocclusion
);
```

Parameters

`directocclusion`

Occlusion factor for a voice for the direct path. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0.

`reverbocclusion`

Occlusion factor for a voice for the reverb mix. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the
**Remarks**

Normally the volume is simply attenuated by the directOcclusion factor. If FMOD_INIT_SOFTWARE_OCCLUSION is specified, **FMOD_SOFTWARE** based sounds will also use frequency filtering, with a very small CPU hit.

**See Also**

- [Channel::get3DOcclusion](#)
- [ChannelGroup::set3DOcclusion](#)
- [FMOD_INITFLAGS](#)
Firelight Technologies FMOD Ex
Channel::set3DPanLevel

Sets how much the 3d engine has an effect on the channel, versus that set by Channel::setPan, Channel::setSpeakerMix, Channel::setSpeakerLevels.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DPanLevel(
    float level
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DPanLevel(
    FMOD_CHANNEL * channel,
    float level
);
```

Parameters

* level

1 = Sound pans and attenuates according to 3d position. 0 = Attenuation is ignored and pan/speaker levels are defined by Channel::setPan, Channel::setSpeakerMix, Channel::setSpeakerLevels. Default = 1 (all by 3D position).

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks
Only affects sounds created **FMOD_3D**.

Useful for morphing a sound between 3D and 2D. This is most common in volumetric sound, when the sound goes from directional, to 'all around you' (and doesn't pan according to listener position/direction). FMOD_INIT_SOFTWARE_HRTF is also interpolated to be 'off' if level = 0, so that you do not get a muffling effect based on location when the sound is supposed to be effectively 2D.

**See Also**

- `Channel::get3DPanLevel`
- `Channel::setSpeakerMix`
- `Channel::setPan`
- `Channel::setSpeakerLevels`
Firelight Technologies FMOD Ex
Channel::set3DSpread

Sets the spread of a 3d sound in speaker space.

Normally a 3d sound is aimed at one position in a speaker array depending on the 3d position, to give it direction. Left and right parts of a stereo sound for example are consequently summed together and become 'mono'.

When increasing the 'spread' of a sound, the left and right parts of a stereo sound rotate away from their original position, to give it more 'stereoness'. The rotation of the sound channels are done in 'speaker space'.

Mono sounds are also able to be 'spread' with this function.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DSpread(
    float angle
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DSpread(
    FMOD_CHANNEL * channel,
    float angle
);
```

Parameters

angle

Speaker spread angle. 0 = all sound channels are located at the same speaker location and is 'mono'. 360 = all subchannels are located at the opposite speaker location to the speaker location that it should be according to 3D position. Default = 0.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the
FMOD_RESULT enumeration.

**Remarks**

Only affects sounds created with **FMODSOFTWARE**.

By default, if a stereo sound was played in 3d, and it was directly in front of you, the left and right part of the stereo sound would be summed into the center speaker (on a 5.1 setup), making it sound mono. This function lets you control the speaker spread of a stereo (and above) sound within the speaker array, to separate the left right part of a stereo sound for example. In the above case, in a 5.1 setup, specifying a spread of 90 degrees would put the left part of the sound in the front left speaker, and the right part of the sound in the front right speaker. This stereo separation remains intact as the listener rotates and the sound moves around the speakers. To summarize (for a stereo sound).

1. A spread angle of 0 makes the stereo sound mono at the point of the 3d emitter.
2. A spread angle of 90 makes the left part of the stereo sound place itself at 45 degrees to the left and the right part 45 degrees to the right.
3. A spread angle of 180 makes the left part of the stereo sound place itself at 90 degrees to the left and the right part 90 degrees to the right.
4. A spread angle of 360 makes the stereo sound mono at the opposite speaker location to where the 3d emitter should be located (by moving the left part 180 degrees left and the right part 180 degrees right). So in this case, behind you when the sound should be in front of you!

Multichannel sounds with channel counts greater than stereo have their sub-channels spread evenly through the specified angle. For example a 6 channel sound over a 90 degree spread has each subchannel located 15 degrees apart from each other in the speaker array.

Mono sounds are spread as if they were a stereo signal, ie the signal is split into 2. The power of the same will remain the same as it spreads around the speakers.
See Also

- `Channel::get3DSpread`
- `FMOD_MODE`
Firelight Technologies FMOD Ex
Channel::setCallback

Sets a callback for a channel for a specific event.

C++ Syntax

```cpp
FMOD_RESULT Channel::setCallback(
    FMOD_CHANNEL_CALLBACK callback
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetCallback(
    FMOD_CHANNEL * channel,
    FMOD_CHANNEL_CALLBACK callback
);
```

Parameters

`callback`

Pointer to a callback to receive the event when it happens.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Currently callbacks are generally driven by `System::update` and will occur when this function is called. The one exception is that `FMOD_CHANNEL_CALLBACKTYPE_END` can also be triggered by a playing channel if `Sound::getSubSound` is called, as it will stop the channel right
at the point of calling the function. Callbacks use the stdcall calling convention on windows. Use F_CALLBACK inbetween your return type and function name, NEVER cast a callback to "FMOD_CHANNEL_CALLBACK" when passing it to this function. This can cause crashes if the callback is incorrectly declared. Here is an example of a channel callback:

```cpp
FMOD_RESULT F_CALLBACK mycallback(FMOD_CHANNEL *channel, FMOD_CHANNEL_CALLBACKTYPE
{   
    FMOD::Channel *cppchannel = (FMOD::Channel *)channel;

    // More code goes here.

    return FMOD_OK;
}
```

**See Also**

- System::update
- FMOD_CHANNEL_CALLBACK
- FMOD_CHANNEL_CALLBACKTYPE
Firelight Technologies FMOD Ex
Channel::setChannelGroup

Sets a channel to belong to a specified channel group. A channel group can contain many channels.

C++ Syntax

```cpp
FMOD_RESULT Channel::setChannelGroup(
    FMOD::ChannelGroup * channelgroup
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetChannelGroup(
    FMOD_CHANNEL * channel,
    FMOD_CHANNELGROUP * channelgroup
);
```

Parameters

`channelgroup`

Pointer to a ChannelGroup object.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Setting a channel to a channel group removes it from any previous group, it does not allow sharing of channel groups.
See Also

- Channel::getChannelGroup
Firelight Technologies FMOD Ex
Channel::setDelay

Sets an end delay for a sound (so that dsp can continue to process the finished sound), set the start of the sound according to the global DSP clock value which represents the time in the mixer timeline.

C++ Syntax

```cpp
FMOD_RESULT Channel::setDelay(
    FMOD_DELAYTYPE delaytype,
    unsigned int delayhi,
    unsigned int delaylo
);```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetDelay(
    FMOD_CHANNEL * channel,
    FMOD_DELAYTYPE delaytype,
    unsigned int delayhi,
    unsigned int delaylo
);```

Parameters

delaytype

See [FMOD_DELAYTYPE](#). This determines what delayhi and delaylo will represent.

delayhi

Top (most significant) 32 bits of a 64bit number representing the time.

delaylo

Bottom (least significant) 32 bits of a 64bit number representing the time.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

**Note!** Only works with sounds created with **FMOD_SOFTWARE**.

Using **FMOD_DELAYTYPE_END_MS**: Setting a delay after a sound ends is sometimes useful to prolong the sound, even though it has stopped, so that DSP effects can trail out, or render the last of their tails. (for example an echo or reverb effect). Remember the delayhi parameter is the only parameter used here, and it is representing milliseconds.

Using **FMOD_DELAYTYPE_DSPCLOCK_START**: **Note!** Works with sounds created with **FMOD_SOFTWARE** only. This allows a sound to be played in the future on an exact sample accurate boundary or DSP clock value. This can be used for synchronizing sounds to start at an exact time in the overall timeline.

What is the 'DSP clock'? The DSP clock represents the output stream to the soundcard, and is incremented by the output rate every second (though of course with much finer granularity than this). So if your output rate is 48khz, the DSP clock will increment by 48000 per second.

The hi and lo values represent this 64bit number, with the delaylo representing the least significant 32bits and the delayhi value representing the most significant 32bits.

Use **FMOD_64BIT_ADD** or **FMOD_64BIT_SUB** to add a hi/lo combination together and cope with wraparound.

See Also

- **FMOD_DELAYTYPE**
- **Channel::getDelay**
• **Channel::isPlaying**
Firelight Technologies FMOD Ex
Channel::setFrequency

Sets the channel's frequency or playback rate, in HZ.

C++ Syntax

```c++
FMOD_RESULT Channel::setFrequency(
    float frequency
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetFrequency(
    FMOD_CHANNEL * channel,
    float frequency
);
```

Parameters

`frequency`

A frequency value in HZ. This value can also be negative to play the sound backwards (negative frequencies allowed with `FMOD_SOFTWARE` based non-stream sounds only).

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

When a sound is played, it plays at the default frequency of the sound which can be set by `Sound::setDefaults`. 
For most file formats, the default frequency is determined by the audio format.

See Also

- Channel::getFrequency
- System::getDriverCaps
- Sound::setDefaults
Channel::setInputChannelMix

Sets the incoming levels in a sound. This means if you have a multichannel sound you can turn channels on and off.
A mono sound has 1 input channel, a stereo has 2, etc. It depends on what type of sound is playing on the channel at the time.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setInputChannelMix(
    float * levels,
    int numlevels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetInputChannelMix(
    FMOD_CHANNEL * channel,
    float * levels,
    int numlevels
);
```

**Parameters**

*levels*

Array of float volume levels, from 0.0 to 1.0. These represent the incoming channels for the sound playing on the channel at the time.

*numlevels*

Number of floats in the array. Maximum = the maximum number of input channels specified in System::setSoftwareFormat.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the
**FMOD_RESULT** enumeration.

**Remarks**

**Note!** This function requires the sound be created with **FMOD_SOFTWARE**. Hardware voices are not capable of this functionality.

**See Also**

- `Channel::getInputChannelMix`
- `Channel::setPan`
- `Channel::setSpeakerMix`
- `Channel::setSpeakerLevels`
- `System::setSoftwareFormat`
Firelight Technologies FMOD Ex
Channel::setLoopCount

Sets a channel to loop a specified number of times before stopping.

C++ Syntax

```cpp
FMOD_RESULT Channel::setLoopCount(
    int loopcount
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetLoopCount(
    FMOD_CHANNEL * channel,
    int loopcount
);
```

Parameters

`loopcount`

Number of times to loop before stopping. 0 = oneshot. 1 = loop once then stop. -1 = loop forever. Default = -1

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function does not affect `FMOD_HARDWARE` based sounds that are not streamable. `FMOD_SOFTWARE` based sounds or any type of sound created with
System::CreateStream or **FMOD_CREATESTREAM** will support this function.

Issues with streamed audio. (Sounds created with with **System::createStream** or **FMOD_CREATESTREAM**). When changing the loop count, sounds created with **System::createStream** or **FMOD_CREATESTREAM** may already have been pre-buffered and executed their loop logic ahead of time, before this call was even made. This is dependant on the size of the sound versus the size of the stream decode buffer. See **FMOD_CREATESOUNDEXINFO**.

If this happens, you may need to flush the stream buffer. To do this, you can call **Channel::setPosition** which forces a flush of the stream buffer. Note this will usually only happen if you have sounds or looppoints that are smaller than the stream decode buffer size. Otherwise you will not normally encounter any problems.

**See Also**

- **Channel::getLoopCount**
- **Channel::setPosition**
- **System::createStream**
- **FMOD_CREATESOUNDEXINFO**
- **FMOD_MODE**
Firelight Technologies FMOD Ex
Channel::setLoopPoints

Sets the loop points within a channel.

C++ Syntax

```c++
FMOD_RESULT Channel::setLoopPoints(
    unsigned int loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int loopend,
    FMOD_TIMEUNIT loopendtype
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetLoopPoints(
    FMOD_CHANNEL * channel,
    unsigned int loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int loopend,
    FMOD_TIMEUNIT loopendtype
);
```

Parameters

**loopstart**

The loop start point. This point in time is played, so it is inclusive.

**loopstarttype**

The time format used for the loop start point. See `FMOD_TIMEUNIT`.

**loopend**

The loop end point. This point in time is played, so it is inclusive.

**loopendtype**

The time format used for the loop end point. See `FMOD_TIMEUNIT`. 
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

Not supported by static sounds created with **FMOD_HARDWARE**. Supported by sounds created with **FMOD_SOFTWARE**, or sounds of any type (hardware or software) created with **System::createStream** or **FMOD_CREATESTREAM**.

If a sound was 44100 samples long and you wanted to loop the whole sound, loopstart would be 0, and loopend would be 44099, not 44100. You wouldn't use milliseconds in this case because they are not sample accurate.

If loop end is smaller or equal to loop start, it will result in an error.
If loop start or loop end is larger than the length of the sound, it will result in an error.

Issues with streamed audio. (Sounds created with **System::createStream** or **FMOD_CREATESTREAM**)

When changing the loop points, sounds created with **System::createStream** or **FMOD_CREATESTREAM** may already have been pre-buffered and executed their loop logic ahead of time, before this call was even made.

This is dependant on the size of the sound versus the size of the stream decode buffer. See **FMOD_CREATESOUNDEXINFO**.

If this happens, you may need to refush the stream buffer. To do this, you can call Channel::setPosition which forces a refush of the stream buffer.

Note this will usually only happen if you have sounds or looppoins that are smaller than the stream decode buffer size. Otherwise you will not normally encounter any problems.

See Also
- **FMOD_TIMEUNIT**
- **FMOD_MODE**
- **Channel::getLoopPoints**
- **Channel::setLoopCount**
- **System::createStream**
- **System::setStreamBufferSize**
- **FMOD_CREATESOUNDEXINFO**
Firelight Technologies FMOD Ex
Channel::setLowPassGain

Sets the gain of the dry signal when lowpass filtering is applied.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setLowPassGain(
    float gain
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetLowPassGain(
    FMOD_CHANNEL * channel,
    float gain
);
```

**Parameters**

*gain*

A linear gain level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [Channel::getLowPassGain](#)
Firelight Technologies FMOD Ex
Channel::setMode

Changes some attributes for a channel based on the mode passed in.

C++ Syntax

```cpp
FMOD_RESULT Channel::setMode(
    FMOD_MODE mode
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetMode(
    FMOD_CHANNEL * channel,
    FMOD_MODE mode
);
```

Parameters

`mode`

Mode bits to set.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Flags supported:
- `FMOD_LOOP_OFF`
- `FMOD_LOOP_NORMAL`
- `FMOD_LOOP_BIDI` (only works with sounds created with
FMOD_SOFTWARE. Otherwise it will behave as FMOD_LOOP_NORMAL
FMOD_3D_HEADRELATIVE
FMOD_3D_WORLDRELATIVE
FMOD_2D (see notes for win32 hardware voices)
FMOD_3D (see notes for win32 hardware voices)
FMOD_3D_INVERSEROLLOFF
FMOD_3D_LINEARROLLOFF
FMOD_3D_LINEARSQUAREROLLOFF
FMOD_3D_CUSTOMROLLOFF
FMOD_3D_IGNOREGEOMETRY
FMOD_DONTRESTOREVIRTUAL

Issues with streamed audio. (Sounds created with with System::createStream or FMOD_CREATESTREAM).
When changing the loop mode, sounds created with System::createStream or FMOD_CREATESTREAM may already have been pre-buffered and executed their loop logic ahead of time, before this call was even made. This is dependant on the size of the sound versus the size of the stream decode buffer. See FMOD_CREATESOUNDEXINFO.
If this happens, you may need to flush the stream buffer. To do this, you can call Channel::setPosition which forces a flush of the stream buffer. Note this will usually only happen if you have sounds or looppoints that are smaller than the stream decode buffer size. Otherwise you will not normally encounter any problems.

Issues with PCM samples. (Sounds created with with System::createSound or FMOD_CREATESAMPLE).
When changing the loop mode, if the sound was set up as FMOD_LOOP_OFF, then set to FMOD_LOOP_NORMAL with this function, the sound may click when playing the end of the sound. This is because the sound needs to be pre-prepared for looping using Sound::setMode, by modifying the content of the pcm data (ie data past the end of the actual sample data) to allow the interpolators to read ahead without clicking. If you use Channel::setMode it will not do this (because different channels may have different loop modes for the same sound) and may click if you try to set it to looping on an unprepared sound. If you want to change the loop mode at runtime it may be better to load the sound as looping first (or use Sound::setMode), to let it pre-prepare the data as if it was looping so that it does not click whenever Channel::setMode is used to turn looping on.
If `FMOD_3D_IGNOREGEOMETRY` is not specified, the flag will be cleared if it was specified previously.

See Also

- `FMOD_MODE`
- `Channel::getMode`
- `Channel::setPosition`
- `Sound::setMode`
- `System::createStream`
- `System::createSound`
- `System::setStreamBufferSize`
- `FMOD_CREATESOUNDEXINFO`
Firelight Technologies FMOD Ex
Channel::setMute

Mutes / un-mutes a channel, effectively silencing it or returning it to its normal volume.

C++ Syntax

```cpp
FMOD_RESULT Channel::setMute(
    bool mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetMute(
    FMOD_CHANNEL * channel,
    FMOD_BOOL mute
);
```

Parameters

`mute`

true = channel becomes muted (silent), false = channel returns to normal volume.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

If a channel belongs to a muted channelgroup, it will stay muted regardless of the channel mute state. The channel mute state will still be reflected internally though, ie `Channel::getMute` will still return the value you set. If the
channelgroup has mute set to false, this function will become effective again.

See Also

- `Channel::getMute`
- `ChannelGroup::setMute`
Firelight Technologies FMOD Ex
Channel::setPan

Sets a channels pan position linearly.

C++ Syntax

```cpp
FMOD_RESULT Channel::setPan(
    float  pan
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetPan(
    FMOD_CHANNEL * channel,
    float  pan
);
```

Parameters

`pan`

A left/right pan level, from -1.0 to 1.0 inclusive. -1.0 = Full left, 0.0 = center, 1.0 = full right. Default = 0.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function only works on sounds created with `FMOD_2D`. 3D sounds are not pannable and will return `FMOD_ERR_NEEDS2D`. 
Only sounds that are mono or stereo can be panned. Multichannel sounds (ie >2 channels) cannot be panned. Mono sounds are panned from left to right using constant power panning (non linear fade). This means when pan = 0.0, the balance for the sound in each speaker is 71% left and 71% right, not 50% left and 50% right. This gives (audibly) smoother pans. Stereo sounds have each left/right value faded up and down according to the specified pan position. This means when pan = 0.0, the balance for the sound in each speaker is 100% left and 100% right. When pan = -1.0, only the left channel of the stereo sound is audible, when pan = 1.0, only the right channel of the stereo sound is audible.

Panning does not work if the speaker mode is **FMOD_SPEAKERMODE_RAW**.

**See Also**

- [Channel::getPan](#)
- [FMOD_SPEAKERMODE](#)
Firelight Technologies FMOD Ex
Channel::setPaused

Sets the paused state of the channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::setPaused(
    bool paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetPaused(
    FMOD_CHANNEL * channel,
    FMOD_BOOL    paused
);
```

Parameters

`paused`

Paused state to set. true = channel is paused. false = channel is unpaused.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

If a channel belongs to a paused channelgroup, it will stay paused regardless of the channel pause state. The channel pause state will still be reflected internally though, ie `Channel::getPaused` will still return the value you set. If the channelgroup has paused set to false, this function will become effective again.
See Also

- Channel::getPaused
- ChannelGroup::setPaused
Firelight Technologies FMOD Ex
Channel::setPosition

Sets the current playback position for the currently playing sound to the specified PCM offset.

C++ Syntax

```cpp
FMOD_RESULT Channel::setPosition(
    unsigned int position,
    FMOD_TIMEUNIT postype
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetPosition(
    FMOD_CHANNEL * channel,
    unsigned int position,
    FMOD_TIMEUNIT postype
);
```

Parameters

- **position**
  Position of the channel to set in units specified in the postype parameter.

- **postype**
  Time unit to set the channel position by. See `FMOD_TIMEUNIT`.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Certain timeunits do not work depending on the file format. For example `FMOD_TIMEUNIT_MODORDER` will not work with an mp3 file.

Note that if you are calling this function on a stream, it has to possibly flush its buffer to get zero latency playback when it resumes playing, therefore it could potentially cause a stall or take a small amount of time to do this.

If you are using `FMOD_NONBLOCKING`, note that a stream will go into `FMOD_OPENSTATE_SETPOSITION` state (see `Sound::getOpenState`) and sound commands will return `FMOD_ERR_NOTREADY`. `Channel::getPosition` will also not update until this non-blocking setposition operation has completed.

Warning! Using a VBR source that does not have an associated seek table or seek information (such as MP3 or MOD/S3M/XM/IT) may cause inaccurate seeking if you specify `FMOD_TIMEUNIT_MS` or `FMOD_TIMEUNIT_PCM`. If you want FMOD to create a pcm vs bytes seek table so that seeking is accurate, you will have to specify `FMOD_ACCURATETIME` when loading or opening the sound. This means there is a slight delay as FMOD scans the whole file when loading the sound to create this table.

See Also

- `Channel::getPosition`
- `FMOD_TIMEUNIT`
- `FMOD_MODE`
- `Sound::getLength`
- `Sound::getOpenState`
Firelight Technologies FMOD Ex
Channel::setPriority

Sets the priority for a channel after it has been played. A sound with a higher priority than another sound will not be stolen or made virtual by that sound.

C++ Syntax

```cpp
FMOD_RESULT Channel::setPriority(
    int priority
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetPriority(
    FMOD_CHANNEL * channel,
    int priority
);
```

**Parameters**

`priority`

Priority for the channel. 0 to 256 inclusive. 0 = most important. 256 = least important. Default = 128.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Priority will make a channel more important or less important than its counterparts. When virtual channels are in place, by default the importance of
the sound (whether it is audible or not when more channels are playing than exist) is based on the volume, or audiblity of the sound. This is determined by distance from the listener in 3d, the volume set with `Channel::setVolume`, channel group volume, and geometry occlusion factors. To make a quiet sound more important, so that it isn't made virtual by louder sounds, you can use this function to increase its importance, and keep it audible.

**See Also**

- `Channel::getPriority`
- `Channel::setVolume`
Firelight Technologies FMOD Ex
Channel::setReverbProperties

Sets the channel specific reverb properties, including things like wet/dry mix.

C++ Syntax

```cpp
FMOD_RESULT Channel::setReverbProperties(
   const FMOD_REVERB_CHANNELPROPERTIES * prop
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetReverbProperties(
   FMOD_CHANNEL * channel,
   const FMOD_REVERB_CHANNELPROPERTIES * prop
);
```

Parameters

*prop*

A pointer to the a **FMOD_REVERB_CHANNELPROPERTIES** structure, with the relevant reverb instance specified in **FMOD_REVERB_CHANNELFLAGS**.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

**Note!** It is important to clear this structure to 0 before calling, or at least specify which reverb instance you are talking about by using the Flags member of **FMOD_REVERB_CHANNELPROPERTIES**, otherwise you may get an
`FMOD_ERR_REVERB_INSTANCE` error.

**See Also**

- `Channel::getReverbProperties`
- `FMOD_REVERB_CHANNELPROPERTIES`
- `FMOD_REVERB_CHANNELFLAGS`
Firelight Technologies FMOD Ex
**Channel::setSpeakerLevels**

Sets the incoming sound levels for a particular speaker. This is a literal setting of channel's internal pan matrix and does NOT perform downmixing depending on speaker mode.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setSpeakerLevels(
    FMOD_SPEAKER speaker,
    float * levels,
    int numlevels
);  
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetSpeakerLevels(
    FMOD_CHANNEL * channel,
    FMOD_SPEAKER speaker,
    float * levels,
    int numlevels

```

**Parameters**

*speaker*

The target speaker to modify the levels for. This can be cast to an integer if you are using `FMOD_SPEAKERMODE_RAW` and want to access up to 15 speakers (output channels).

*levels*

An array of floating point numbers from 0.0 to 1.0 representing the volume of each input channel of a sound. See remarks for more.

*numlevels*

The number of floats within the levels parameter being passed to this function. In the case of the above mono or stereo sound, 1 or 2 could be used respectively.
If the sound being played was an 8 channel multichannel sound then 8 levels would be used.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

As an example of usage of this function, if the sound played on this speaker was mono, only 1 level would be needed. If the sound played on this channel was stereo, then an array of 2 floats could be specified. For example `{ 0, 1 }` on a channel playing a stereo sound would mute the left part of the stereo sound when it is played on this speaker.

Only speakers that are usable with the current speaker mode will be accepted. Anything else will return `FMOD_ERR_INVALID_SPEAKER`.

When using `FMOD_SPEAKERMODE_MONO` it is preferable to use the alias `FMOD_SPEAKER_MONO` (or index 0), as `FMOD_SPEAKER_FRONT_LEFT` doesn't really make sense in that setting. When using `FMOD_SPEAKERMODE_RAW`, the 'speaker' parameter can be cast to an integer and used as a raw speaker index, disregarding FMOD's speaker mappings.

**Warning.** This function will allocate memory for the speaker level matrix and attach it to the channel. If you prefer not to have a dynamic memory allocation done at this point use `Channel::setSpeakerMix` instead.

**See Also**

- `Channel::getSpeakerLevels`
- Channel::setSpeakerMix
- FMOD_SPEAKERMODE
- FMOD_SPEAKER
- FMOD_OUTPUTTYPE
Firelight Technologies FMOD Ex
Channel::setSpeakerMix

Sets the channel's speaker volume levels for each speaker individually.

C++ Syntax

```cpp
FMOD_RESULT Channel::setSpeakerMix(
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float backleft,
    float backright,
    float sideleft,
    float sideright
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetSpeakerMix(
    FMOD_CHANNEL * channel,
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float backleft,
    float backright,
    float sideleft,
    float sideright
);
```

Parameters

`frontleft`

Volume level for this channel in the front left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = normal volume, 5.0 = 5x amplification.

`frontright`

Volume level for this channel in the front right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = normal volume, up to 5.0 = 5x amplification.
**center**

Volume level for this channel in the center speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = normal volume, 5.0 = 5x amplification.

**lfe**

Volume level for this channel in the subwoofer speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = normal volume, 5.0 = 5x amplification.

**backleft**

Volume level for this channel in the back left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = normal volume, 5.0 = 5x amplification.

**backright**

Volume level for this channel in the back right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = normal volume, 5.0 = 5x amplification.

**sideleft**

Volume level for this channel in the side left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = normal volume, 5.0 = 5x amplification.

**sideright**

Volume level for this channel in the side right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = normal volume, 5.0 = 5x amplification.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**
This function only fully works on sounds created with **FMOD_2D** and **FMOD_SOFTWARE**. **FMOD_3D** based sounds only allow setting of LFE channel, as all other speaker levels are calculated by FMOD's 3D engine.

Speakers specified that don't exist will simply be ignored.

For more advanced speaker control, including sending the different channels of a stereo sound to arbitrary speakers, see [Channel::setSpeakerLevels](#).

This function allows amplification! You can go up to 5 times the volume of a normal sound, but warning this may cause clipping/distortion! Useful for LFE boosting.

**See Also**

- [Channel::getSpeakerMix](#)
- [Channel::setSpeakerLevels](#)
- [FMOD_SPEAKERMODE](#)
Firelight Technologies FMOD Ex
Channel::setUserData

Sets a user value that the Channel object will store internally. Can be retrieved with Channel::getUserData.

C++ Syntax

```cpp
FMOD_RESULT Channel::setUserData(  
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetUserData(
    FMOD_CHANNEL * channel,  
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the Channel object.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using `Channel::getUserData` would help in the identification of the object.

NOTE: If this channel was spawned by the event system then its user data field will be set, by the event system, to the event instance handle that spawned it and this function should NOT be called.

**See Also**

- `Channel::getUserData`

Version 4.44.07 Built on Feb 11, 2013
Channel::setVolume

Sets the volume for the channel linearly.

C++ Syntax

```cpp
FMOD_RESULT Channel::setVolume(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetVolume(
    FMOD_CHANNEL * channel,
    float volume
);
```

Parameters

`volume`

A linear volume level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

When a sound is played, it plays at the default volume of the sound which can be set by `Sound::setDefaults`. For most file formats, the volume is determined by the audio format.
See Also

- `Channel::getVolume`
- `ChannelGroup::setVolume`
- `Sound::setDefaults`
Firelight Technologies FMOD Ex
Channel::stop

Stops the channel from playing. Makes it available for re-use by the priority system.

C++ Syntax

```cpp
FMOD_RESULT Channel::stop();
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Stop(FMOD_CHANNEL * channel);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `System::playSound`
Firelight Technologies FMOD Ex
ChannelGroup Interface

ChannelGroup::addDSP ChannelGroup::addGroup
ChannelGroup::get3DOcclusion
ChannelGroup::getChannel
ChannelGroup::getDSPHead
ChannelGroup::getGroup
ChannelGroup::getMemoryInfo
ChannelGroup::getMute
ChannelGroup::getName
ChannelGroup::getNumChannels
ChannelGroup::getNumGroups
ChannelGroup::getParentGroup
ChannelGroup::getPaused
ChannelGroup::getPitch
ChannelGroup::getSpectrum
ChannelGroup::getSystemObject
ChannelGroup::getUserData
ChannelGroup::getVolume
ChannelGroup::getWaveData
ChannelGroup::override3DAttributes
ChannelGroup::overrideFrequency
ChannelGroup::overridePan
ChannelGroup::overrideReverbProperties
ChannelGroup::overrideSpeakerMix
ChannelGroup::overrideVolume
ChannelGroup::release
ChannelGroup::set3DOcclusion
ChannelGroup::setMute
ChannelGroup::setPaused
ChannelGroup::setPitch
ChannelGroup::setUserData
ChannelGroup::setVolume
ChannelGroup::stop
Firelight Technologies FMOD Ex
ChannelGroup::addDSP

Adds a DSP effect to this channelgroup, affecting all channels that belong to it. Because it is a submix, only one instance of the effect is added, and all subsequent channels are affected.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::addDSP(
    FMOD::DSP * dsp,
    FMOD::DSPConnection ** connection
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_AddDSP(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_DSP * dsp,
    FMOD_DSPCONNECTION ** connection
);
```

Parameters

dsp

Pointer to the dsp effect to add. This can be created with `System::createDSP`, `System::createDSPByType`, `System::createDSPByPlugin`.

connection

A pointer to the connection involved between the ChannelGroup DSP head and the specified dsp unit. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

This function is a wrapper function to insert a DSP unit at the top of the channel group DSP chain. It disconnects the head unit from its input, then inserts the unit at the head and reconnects the previously disconnected input back as as an input to the new unit.

**Note:** The connection pointer retrieved here will become invalid if you disconnect the 2 dsp units that use it.

**Note:** If this function returns `FMOD_ERR_DSP_NOTFOUND` when called on an event's channelgroup, the channelgroup's DSP unit may have been optimized away by the event system. Use `FMOD_EVENT_USERDSP` when getting the event to force the channelgroup to contain a DSP unit.

See Also

- `ChannelGroup::getDSPHead`
- `System::createDSP`
- `System::createDSPByType`
- `System::createDSPByPlugin`
- `System::getMasterChannelGroup`
- `System::createChannelGroup`
- `System::addDSP`
- `Channel::addDSP`
- `DSP::remove`
Firelight Technologies FMOD Ex
ChannelGroup::addGroup

Adds a channel group as a child of the current channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::addGroup(
    FMOD::ChannelGroup * group
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_AddGroup(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_CHANNELGROUP * group
);
```

Parameters

group

channel group to add as a child.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- ChannelGroup::getNumGroups
- ChannelGroup::getGroup
Firelight Technologies FMOD Ex
ChannelGroup::get3DOcclusion

Retrieves the master occlusion factors for the channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::get3DOcclusion(
    float * directocclusion,
    float * reverbocclusion
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_Get3DOcclusion(
    FMOD_CHANNELGROUP * channelgroup,
    float * directocclusion,
    float * reverbocclusion
);
```

Parameters

**directocclusion**

Address of a variable that receives the occlusion factor for the direct path. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0. Optional. Specify 0 or NULL to ignore.

**reverbocclusion**

Address of a variable that receives the occlusion factor for the reverb mix. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
See Also

- `ChannelGroup::set3DOcclusion`
- `Channel::set3DOcclusion`
- `Channel::get3DOcclusion`
- `System::getMasterChannelGroup`
Firelight Technologies FMOD Ex
ChannelGroup::getChannel

Retrieves the handle to a channel from the current channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getChannel(
    int index,
    FMOD::Channel ** channel
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetChannel(
    FMOD_CHANNELGROUP * channelgroup,
    int index,
    FMOD_CHANNEL ** channel
);
```

Parameters

index

Index of the channel inside the channel group, from 0 to the number of channels returned by `ChannelGroup::getNumChannels`.

channel

Address of a variable to receive a pointer to a Channel object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- ChannelGroup::getNumChannels
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::getDSPHead

Retrieves the DSP unit responsible for this channel group. When channels are submixed to this channel group, this is the DSP unit they target.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getDSPHead(
    FMOD::DSP ** dsp
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetDSPHead(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_DSP ** dsp
);
```

Parameters

`dsp`

Address of a variable to receive the pointer to the head DSP unit for this channel group.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Use this unit if you wish to connect custom DSP units to the channelgroup or filter the channels in the channel group by inserting filter units between this one
and the incoming channel mixer unit.
Read the tutorial on DSP if you wish to know more about this. It is not recommended using this if you do not understand how the FMOD Ex DSP network is connected.
Alternatively you can simply add effects by using `ChannelGroup::addDSP` which does the connection / disconnection work for you.

**Note:** If this function returns `FMOD_ERR_DSP_NOTFOUND` when called on an event's channelgroup, the channelgroup's DSP unit may have been optimized away by the event system. Use `FMOD_EVENT_USERDSP` when getting the event to force the channelgroup to contain a DSP unit.

**See Also**

- `ChannelGroup::addDSP`
- `System::createDSP`
- `System::createDSPByType`
- `System::createDSPByPlugin`
- `System::getMasterChannelGroup`
- `System::createChannelGroup`
ChannelGroup::getGroup

Retrieves a handle to a specified sub channelgroup.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getGroup(
    int index,
    FMOD::ChannelGroup ** group
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetGroup(
    FMOD_CHANNELGROUP * channelgroup,
    int index,
    FMOD_CHANNELGROUP ** group
);
```

Parameters

`index`

Index to specify which sub channelgroup to receive.

`group`

Address of a variable to receive a pointer to a channelgroup.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also
• `ChannelGroup::getNumGroups`
• `ChannelGroup::getParentGroup`
• `ChannelGroup::addGroup`
Firelight Technologies FMOD Ex
ChannelGroup::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetMemoryInfo(
    FMOD_CHANNELGROUP * channelgroup,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits
Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

event_memorybits
Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

memoryused
Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated \texttt{FMOD\_MEMORY\_USAGE\_DETAILS} structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is \texttt{FMOD\_OK}.
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

**Remarks**

See \texttt{System::getMemoryInfo} for more details.

**See Also**

- \texttt{FMOD\_MEMBITS}
- \texttt{FMOD\_EVENT\_MEMBITS}
- \texttt{FMOD\_MEMORY\_USAGE\_DETAILS}
- \texttt{System::getMemoryInfo}
ChannelGroup::getMute

Retrieves the mute state of a ChannelGroup.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getMute(
    bool * mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetMute(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_BOOL * mute
);
```

Parameters

`mute`

Address of a variable to receive the pause state of the channelgroup.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `ChannelGroup::setMute`
- `Channel::setMute`
- `Channel::getMute`
- `System::getMasterChannelGroup`
- `System::createChannelGroup`
Firelight Technologies FMOD Ex
ChannelGroup::getName

Retrieves the name of the channel group. The name is set when the group is created.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getName(
    char * name,
    int namelen
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetName(
    FMOD_CHANNELGROUP * channelgroup,
    char * name,
    int namelen
);
```

Parameters

**name**

Address of a variable that receives the name of the channel group.

**namelen**

Length in bytes of the target buffer to receive the string.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
See Also

- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::getNumChannels

Retrieves the current number of assigned channels to this channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getNumChannels(
    int * numchannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetNumChannels(
    FMOD_CHANNELGROUP * channelgroup,
    int * numchannels
);
```

Parameters

`numchannels`

Address of a variable to receive the current number of assigned channels in this channel group.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Use this function to enumerate the channels within the channel group. You can then use `ChannelGroup::getChannel` to retrieve each individual channel.
See Also

- ChannelGroup::getChannel
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::getNumGroups

Retrieves the number of sub groups under this channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getNumGroups(
    int * numgroups
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetNumGroups(
    FMOD_CHANNELGROUP * channelgroup,
    int * numgroups
);
```

Parameters

`numgroups`
Address of a variable to receive the number of channel groups within this channel group.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- ChannelGroup::getGroup
- ChannelGroup::addGroup
Firelight Technologies FMOD Ex
ChannelGroup::getParentGroup

Retrieves a handle to this channelgroup's parent channelgroup.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getParentGroup(
    FMOD::ChannelGroup ** group
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetParentGroup(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_CHANNELGROUP ** group
);
```

Parameters

`group`

Address of a variable to receive a pointer to a channelgroup.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- ChannelGroup::getNumGroups
- ChannelGroup::getGroup
Firelight Technologies FMOD Ex
ChannelGroup::getPaused

Retrieves the pause state of a ChannelGroup.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getPaused(
    bool * paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetPaused(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_BOOL * paused
);
```

Parameters

* paused

Address of a variable to receive the pause state of the channelgroup.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- ChannelGroup::setPaused
- Channel::setPaused
- Channel::getPaused
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::getPitch

Retrieves the master pitch level for the channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getPitch(
    float * pitch
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetPitch(
    FMOD_CHANNELGROUP * channelgroup,
    float * pitch
);
```

Parameters

`pitch`

Address of a variable to receive the channel group pitch value, from 0.0 to 10.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- ChannelGroup::setPitch
- ChannelGroup::overrideFrequency
- System::getMasterChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::getSpectrum

Retrieves the spectrum from the currently playing channels assigned to this channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getSpectrum(
    float * spectrumarray,
    int numvalues,
    int channeloffset,
    FMOD_DSP_FFT_WINDOW windowtype
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetSpectrum(
    FMOD_CHANNELGROUP * channelgroup,
    float * spectrumarray,
    int numvalues,
    int channeloffset,
    FMOD_DSP_FFT_WINDOW windowtype
);
```

Parameters

**spectrumarray**

Address of a variable that receives the spectrum data. This is an array of floating point values. Data will range is 0.0 to 1.0. Decibels = 10.0f * (float)log10(val) * 2.0f; See remarks for what the data represents.

**numvalues**

Size of array in floating point values being passed to the function. Must be a power of 2. (ie 128/256/512 etc). Min = 64. Max = 8192.

**channeloffset**

Channel of the signal to analyze. If the signal is multichannel (such as a stereo
output), then this value represents which channel to analyze. On a stereo signal 0 = left, 1 = right.

windowtype

"Pre-FFT" window method. This filters the PCM data before entering the spectrum analyzer to reduce transient frequency error for more accurate results. See FMOD_DSP_FFT_WINDOW for different types of fft window techniques possible and for a more detailed explanation.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

The larger the numvalues, the more CPU the FFT will take. Choose the right value to trade off between accuracy / speed. The larger the numvalues, the more 'lag' the spectrum will seem to inherit. This is because the FFT window size stretches the analysis back in time to what was already played. For example if the window size happened to be 44100 and the output rate was 44100 it would be analyzing the past second of data, and giving you the average spectrum over that time period. If you are not displaying the result in dB, then the data may seem smaller than it should be. To display it you may want to normalize the data - that is, find the maximum value in the resulting spectrum, and scale all values in the array by 1 / max. (ie if the max was 0.5f, then it would become 1). To get the spectrum for both channels of a stereo signal, call this function twice, once with channeloffset = 0, and again with channeloffset = 1. Then add the spectrums together and divide by 2 to get the average spectrum for both channels.

What the data represents.
To work out what each entry in the array represents, use this formula
entry_hz = (output_rate / 2) / numvalues

The array represents amplitudes of each frequency band from 0hz to the nyquist rate. The nyquist rate is equal to the output rate divided by 2.
For example when FMOD is set to 44100hz output, the range of represented frequencies will be 0hz to 22049hz, a total of 22050hz represented.
If in the same example, 1024 was passed to this function as the numvalues, each entry's contribution would be as follows.

entry_hz = (44100 / 2) / 1024
entry_hz = 21.53 hz

Note: This function only displays data for sounds playing that were created with FMOD_SOFTWARE. FMOD_HARDWARE based sounds are played using the sound card driver and are not accessible.
With the Event system, events are now optimized to remove the DSP unit from a channelgroup to save memory. This will result in the error FMOD_ERR_DSP_NOTFOUND. If you wish to force the DSP node to be created with an event, use FMOD_EVENT_USERDSP flag when getting an event.

See Also

- FMOD_DSP_FFT_WINDOW
- System::getSpectrum
- Channel::getSpectrum
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::getSystemObject

Retrieves the parent System object that created this channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getSystemObject(
    FMOD::System ** system
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetSystemObject(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_SYSTEM ** system
);
```

Parameters

`system`

Address of a variable that receives the System object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [System::createChannelGroup](#)
- [System::getMasterChannelGroup](#)
Firelight Technologies FMOD Ex
ChannelGroup::getUserData

Retrieves the user value that that was set by calling the ChannelGroup::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetUserData(
    FMOD_CHANNELGROUP * channelgroup,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the to user data specified with the ChannelGroup::setUserData function.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- ChannelGroup::setUserData
Firelight Technologies FMOD Ex
ChannelGroup::getVolume

Retrieves the master volume level for the channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getVolume(
    float * volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetVolume(
    FMOD_CHANNELGROUP * channelgroup,
    float * volume
);
```

Parameters

`volume`

Address of a variable to receive the channel group volume level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- ChannelGroup::setVolume
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::getWaveData

Retrieves a pointer to a block of PCM data that represents the currently playing waveform for this channel group.
This function is useful for a very easy way to plot an oscilliscope.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getWaveData(
    float * wavearray,
    int numvalues,
    int channeloffset
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetWaveData(
    FMOD_CHANNELGROUP * channelgroup,
    float * wavearray,
    int numvalues,
    int channeloffset
);
```

Parameters

wavearray
Address of a variable that receives the currently playing waveform data. This is an array of floating point values.

numvalues
Number of floats to write to the array. Maximum value = 16384.

channeloffset
Offset into multichannel data. Mono channels use 0. Stereo channels use 0 = left, 1 = right. More than stereo use the appropriate index.
Return Values

If the function succeeds then the return value is \texttt{FMOD\_OK}.
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

Remarks

This is the actual resampled, filtered and volume scaled data, at the time this function is called.

Do not use this function to try and display the whole waveform of the sound, as this is more of a 'snapshot' of the current waveform at the time it is called, and could return the same data if it is called very quickly in succession.
See the DSP API to capture a continual stream of wave data as it plays, or see \texttt{Sound::lock} / \texttt{Sound::unlock} if you want to simply display the waveform of a sound.

This function allows retrieval of left and right data for a stereo sound individually. To combine them into one signal, simply add the entries of each separate buffer together and then divide them by 2.

\textbf{Note:} This function only displays data for sounds playing that were created with \texttt{FMOD\_SOFTWARE}, \texttt{FMOD\_HARDWARE} based sounds are played using the sound card driver and are not accessible.
With the Event system, events are now optimized to remove the DSP unit from a channelgroup to save memory. This will result in the error \texttt{FMOD\_ERR\_DSP\_NOTFOUND}. If you wish to force the DSP node to be created with an event, use \texttt{FMOD\_EVENT\_USERDSP} flag when getting an event.

See Also

- \texttt{System::getMasterChannelGroup}
- \texttt{System::createChannelGroup}
- \texttt{Sound::lock}
- \texttt{Sound::unlock}
Firelight Technologies FMOD Ex
ChannelGroup::override3DAttributes

Overrides the position and velocity of all channels within this channel group and those of any sub channelgroups.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::override3DAttributes(
    const FMOD_VECTOR * pos,
    const FMOD_VECTOR * vel
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_Override3DAttributes(
    FMOD_CHANNELGROUP * channelgroup,
    const FMOD_VECTOR * pos,
    const FMOD_VECTOR * vel
);
```

**Parameters**

*pos*

Position in 3D space of the channels in the group. Specifying 0 / null will ignore this parameter.

*vel*

Velocity in 'distance units per second' in 3D space of the group of channels. See remarks. Specifying 0 / null will ignore this parameter.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

A 'distance unit' is specified by System::set3DSettings. By default this is set to meters which is a distance scale of 1.0.

See Also

- Channel::set3DAttributes
- Channel::get3DAttributes
- FMOD_VECTOR
- System::set3DSettings
Firelight Technologies FMOD Ex
ChannelGroup::overrideFrequency

Overrides the frequency or playback rate, in HZ of all channels within this channel group and those of any sub channel groups.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::overrideFrequency(
    float frequency
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_OverrideFrequency(
    FMOD_CHANNELGROUP * channelgroup,
    float frequency
);
```

Parameters

frequency

A frequency value in HZ. This value can also be negative to play the sound backwards (negative frequencies allowed with FMODSOFTWARE based non-stream sounds sounds only).

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

When a sound is played, it plays at the default frequency of the sound which can
be set by Sound::setDefaults.
For most file formats, the default frequency is determined by the audio format.

See Also

- Channel::setFrequency
- Channel::getFrequency
- System::getDriverCaps
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::overridePan

Sets pan position linearly of all channels within this channel group and those of any sub channelgroups.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::overridePan(
    float pan
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_OverridePan(
    FMOD_CHANNELGROUP * channelgroup,
    float pan
);
```

**Parameters**

*pan*

A left/right pan level, from -1.0 to 1.0 inclusive. -1.0 = Full left, 0.0 = center, 1.0 = full right. Default = 0.0.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Panning only works on sounds created with `FMOD_2D`. 3D sounds are not pannable.
Only sounds that are mono or stereo can be panned. Multichannel sounds (ie >2 channels) cannot be panned.

Mono sounds are panned from left to right using constant power panning. This means when pan = 0.0, the balance for the sound in each speaker is 71% left and 71% right, not 50% left and 50% right. This gives (audibly) smoother pans. Stereo sounds have each left/right value faded up and down according to the specified pan position. This means when pan = 0.0, the balance for the sound in each speaker is 100% left and 100% right. When pan = -1.0, only the left channel of the stereo sound is audible, when pan = 1.0, only the right channel of the stereo sound is audible.

See Also

- System::getMasterChannelGroup
- System::createChannelGroup
- Channel::setPan
- Channel::getPan
ChannelGroup::overrideReverbProperties

Overrides the reverb properties of all channels within this channel group and those of any sub channelgroups.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::overrideReverbProperties(
    const FMOD_REVERB_CHANNELPROPERTIES * prop
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_OverrideReverbProperties(
    FMOD_CHANNELGROUP * channelgroup,
    const FMOD_REVERB_CHANNELPROPERTIES * prop
);
```

Parameters

`prop`

Pointer to a `FMOD_REVERB_CHANNELPROPERTIES` structure definition.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `FMOD_REVERB_CHANNELPROPERTIES`
- `System::setReverbProperties`
- `System::getReverbProperties`
• **Channel::setReverbProperties**
• **Channel::getReverbProperties**
• **System::getMasterChannelGroup**
• **System::createChannelGroup**
Firelight Technologies FMOD Ex
ChannelGroup::overrideSpeakerMix

Override all channel speaker levels for each speaker individually.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::overrideSpeakerMix(
  float frontleft,
  float frontright,
  float center,
  float lfe,
  float backleft,
  float backright,
  float sideleft,
  float sideright
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_OverrideSpeakerMix(
  FMOD_CHANNELGROUP * channelgroup,
  float frontleft,
  float frontright,
  float center,
  float lfe,
  float backleft,
  float backright,
  float sideleft,
  float sideright
);
```

**Parameters**

*frontleft*

Level for this channel in the front left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume.

*frontright*

Level for this channel in the front right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume.
center
Level for this channel in the center speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume.

lfe
Level for this channel in the subwoofer speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume.

backleft
Level for this channel in the back left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume.

backright
Level for this channel in the back right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume.

sideleft
Level for this channel in the side left speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume.

sideright
Level for this channel in the side right speaker of a multichannel speaker setup. 0.0 = silent, 1.0 = full volume.

Return Values
If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks
This function only works on sounds created with **FMOD_2D**. 3D sounds are not pannable and will return **FMOD_ERR_NEEDS2D**.

Only sounds create with **FMOD_SOFTWARE** playing on this channel will allow this functionality.

Speakers specified that don't exist will simply be ignored.

For more advanced speaker control, including sending the different channels of a stereo sound to arbitrary speakers, see **Channel::setSpeakerLevels**.

**See Also**

- [Channel::setSpeakerMix](#)
- [Channel::getSpeakerMix](#)
- [Channel::setSpeakerLevels](#)
Firelight Technologies FMOD Ex
ChannelGroup::overrideVolume

Overrides the volume of all channels within this channel group and those of any sub channelgroups.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::overrideVolume(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_OverrideVolume(
    FMOD_CHANNELGROUP * channelgroup,
    float volume
);
```

Parameters

`volume`

A linear volume level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
This is not to be used as a master volume for the group, as it will modify the volumes of the channels themselves.

If you want to scale the volume of the group, use `ChannelGroup::setVolume`. 
See Also

- `System::getMasterChannelGroup`
- `System::createChannelGroup`
- `ChannelGroup::setVolume`
Firelight Technologies FMOD Ex
ChannelGroup::release

Frees a channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::release();
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_Release(FMOD_CHANNELGROUP * channelgr
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

All channels assigned to this group are returned back to the master channel group owned by the System object. See `System::getMasterChannelGroup`.
All child groups assigned to this group are returned back to the master channel group owned by the System object. See `System::getMasterChannelGroup`.

See Also

- `System::createChannelGroup`
- `System::getMasterChannelGroup`
Firelight Technologies FMOD Ex
ChannelGroup::set3DOcclusion

Sets the master occlusion factors for the channel group.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::set3DOcclusion(
    float directocclusion,
    float reverbocclusion
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Set3DOcclusion(
    FMOD_CHANNELGROUP * channelgroup,
    float directocclusion,
    float reverbocclusion
);
```

**Parameters**

*directocclusion*

Occlusion factor for the direct path. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0.

*reverbocclusion*

Occlusion factor for the reverb mix. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

This function does not go through and overwrite the channel occlusion factors. It scales them by the channel group's occlusion factors. That way when Channel::set3DOcclusion / Channel::get3DOcclusion is called the respective individual channel occlusion factors will still be preserved. This means that final Channel occlusion values will be affected by both ChannelGroup occlusion and geometry (if any).

See Also

- ChannelGroup::get3DOcclusion
- Channel::set3DOcclusion
- Channel::get3DOcclusion
- System::getMasterChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::setMute

Mutes a channelgroup, and the channels within it, or unmutes any unmuted channels if set to false.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setMute(
    bool mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetMute(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_BOOL mute
);
```

Parameters

mute

Mute state to set. true = channelgroup state is set to muted. false = channelgroup state is set to unmuted.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

A channelgroup maintains a mute state, that affects channelgroups and channels within it. If a channelgroup is muted, all channelgroups and channels below it
will become muted. Channels will not have their per channel mute state overwritten, so that when a channel group is unmuted, the muted state of the channels will correct as they were set on a per channel basis. This means even though a channel is muted, it can return false when you call \texttt{Channel::getMute} on that channel, because that was the state of the channel at the time before the ChannelGroup was muted.

\textbf{See Also}

- \texttt{ChannelGroup::getMute}
- \texttt{Channel::setMute}
- \texttt{Channel::getMute}
- \texttt{System::getMasterChannelGroup}
- \texttt{System::createChannelGroup}
Firelight Technologies FMOD Ex
ChannelGroup::setPaused

Pauses a channelgroup, and the channels within it, or unpauses any unpau sed channels if set to false.

C++ Syntax

FMOD_RESULT ChannelGroup::setPaused(
    bool paused
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_SetPaused(
    FMOD_CHANNELGROUP * channelgroup,
    FMOD_BOOL paused
);

Parameters

paused

Paused state to set. true = channelgroup state is set to paused. false = channelgroup state is set to unpaused.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

A channelgroup maintains a paused state, that affects channelgroups and channels within it. If a channelgroup is paused, all channelgroups and channels
below it will become paused. Channels will not have their per channel pause state overwritten, so that when a channelgroup is unpued, the paused state of the channels will correct as they were set on a per channel basis. This means even though a channel is paused, it can return false when you call Channel::getPaused on that channel, because that was the state of the channel at the time before the ChannelGroup was paused.

See Also

- ChannelGroup::getPaused
- Channel::setPaused
- Channel::getPaused
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::setPitch

Sets the master pitch for the channel group.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::setPitch(
    float pitch
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_SetPitch(
    FMOD_CHANNELGROUP * channelgroup,
    float pitch
);
```

**Parameters**

*`pitch`*

A pitch level, from 0.0 to 10.0 inclusive. 0.5 = half pitch, 2.0 = double pitch. Default = 1.0.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This function does not go through and overwrite the channel frequencies. It scales them by the channel group's pitch. That way when `Channel::setFrequency` / `Channel::getFrequency` is called the
respective individual channel frequencies will still be preserved.

See Also

- ChannelGroup::overrideFrequency
- System::getMasterChannelGroup
- ChannelGroup::getPitch
- Channel::setFrequency
- Channel::getFrequency
- ChannelGroup::overrideFrequency
Firelight Technologies FMOD Ex
**ChannelGroup::setUserData**

Sets a user value that the ChannelGroup object will store internally. Can be retrieved with `ChannelGroup::getUserData`.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::setUserData(
    void * userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_SetUserData(
    FMOD_CHANNELGROUP * channelgroup,
    void * userdata
);
```

**Parameters**

`userdata`

Address of user data that the user wishes stored within the ChannelGroup object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This function is primarily used in case the user wishes to 'attach' data to an FMOD object. It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using ChannelGroup::getUserData would help in the identification of the object.

See Also

- ChannelGroup::getUserData
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
ChannelGroup::setVolume

Sets the master volume for the channel group linearly.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setVolume(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetVolume(
    FMOD_CHANNELGROUP * channelgroup,
    float volume
);
```

Parameters

`volume`

A linear volume level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function does not go through and overwrite the channel volumes. It scales them by the channel group's volume. That way when `Channel::setVolume / Channel::getVolume` is called the
respective individual channel volumes will still be preserved.

See Also

- ChannelGroup::setVolume
- Channel::setVolume
- Channel::getVolume
- ChannelGroup::overrideVolume
Firelight Technologies FMOD Ex
ChannelGroup::stop

Stops all channels within the channelgroup.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::stop();
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_Stop(FMOD_CHANNELGROUP * channelgroup
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- System::playSound
- System::getMasterChannelGroup
- System::createChannelGroup
Firelight Technologies FMOD Ex
SoundGroup Interface

SoundGroup::getMaxAudible  SoundGroup::getMaxAudibleBehavior
SoundGroup::getMemoryInfo
SoundGroup::getMuteFadeSpeed
SoundGroup::getName
SoundGroup::getNumPlaying
SoundGroup::getNumSounds
SoundGroup::getSound
SoundGroup::getSystemObject
SoundGroup::getUserData
SoundGroup::getVolume
SoundGroup::release
SoundGroup::setMaxAudible
SoundGroup::setMaxAudibleBehavior
SoundGroup::setMuteFadeSpeed
SoundGroup::setUserData
SoundGroup::setVolume
SoundGroup::stop
Firelight Technologies FMOD Ex
SoundGroup::getMaxAudible

Retrieves the number of concurrent playbacks of sounds in a sound group to the specified value. If the sounds in the sound group are playing this many times, any attempts to play more of the sounds in the sound group will fail with FMOD_ERR_MAXAUDIBLE.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getMaxAudible(  
    int * maxaudible
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetMaxAudible(  
    FMOD_SOUNDDGROUP * soundgroup,  
    int * maxaudible
);
```

Parameters

`maxaudible`

Address of a variable to receive the number of playbacks to be audible at once. -1 = unlimited. 0 means no sounds in this group will succeed. Default = -1.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks
**SoundGroup::getNumPlaying** can be used to determine how many instances of the sounds in the sound group are playing.

**See Also**

- **SoundGroup::setMaxAudible**
- **SoundGroup::getNumPlaying**
- **System::createSoundGroup**
- **System::getMasterSoundGroup**
Firelight Technologies FMOD Ex
SoundGroup::getMaxAudibleBehavior

Retrieves the current max audible behavior method.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getMaxAudibleBehavior(
    FMOD_SOUNDGROUP_BEHAVIOR * behavior
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetMaxAudibleBehavior(
    FMOD_SOUNDGROUP * soundgroup,
    FMOD_SOUNDGROUP_BEHAVIOR * behavior
);
```

Parameters

`behavior`

Address of a variable to recieve the current sound group max playbacks behavior. Default is `FMOD_SOUNDGROUP_BEHAVIOR_FAIL`.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `FMOD_SOUNDGROUP_BEHAVIOR`
- `SoundGroup::setMaxAudibleBehavior`
- `SoundGroup::setMaxAudible`
- SoundGroup::getMaxAudible
- SoundGroup::setMuteFadeSpeed
- SoundGroup::getMuteFadeSpeed
- System::createSoundGroup
- System::getMasterSoundGroup

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
SoundGroup::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetMemoryInfo(
    FMOD_SOUNDGROUP * soundgroup,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

**memorybits**

Memory usage bits for FMOD Ex. See [FMOD_MEMBITS](#).

**event_memorybits**

Memory usage bits for FMOD Event System. See [FMOD_EVENT_MEMBITS](#).

**memoryused**

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

**memoryused_details**
Optional. Specify 0 to ignore. Address of a user-allocated `FMOD_MEMORY_USAGE_DETAILS` structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See [System::getMemoryInfo](#) for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGE_DETAILS`
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
SoundGroup::getMuteFadeSpeed

Retrieves the current time in seconds for
FMOD_SOUNDGROUP_BEHAVIOR_MUTE behavior to fade with.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getMuteFadeSpeed(
    float * speed
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetMuteFadeSpeed(
    FMOD_SOUNDGROUP * soundgroup,
    float * speed
);
```

Parameters

`speed`

Address of a variable to receive the fade time in seconds (1.0 = 1 second).
Default = 0.0. (no fade).

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the
FMOD_RESULT enumeration.

Remarks

If a mode besides FMOD_SOUNDGROUP_BEHAVIOR_MUTE is used, the fade speed is ignored.
See Also

- `SoundGroup::setMuteFadeSpeed`
- `SoundGroup::setMaxAudibleBehavior`
- `SoundGroup::getMaxAudibleBehavior`
- `SoundGroup::setMaxAudible`
- `SoundGroup::getMaxAudible`
- `System::createSoundGroup`
- `System::getMasterSoundGroup`
Firelight Technologies FMOD Ex
SoundGroup::getName

Retrieves the name of the sound group.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getName(
    char * name,
    int namelen
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetName(
    FMOD_SOUNDGROUP * soundgroup,
    char * name,
    int namelen
);
```

Parameters

name

Address of a variable that receives the name of the sound group.

namelen

Length in bytes of the target buffer to receive the string.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also
- **System::createSoundGroup**
- **System::getMasterSoundGroup**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
SoundGroup::getNumPlaying

Retrieves the number of currently playing channels for the sound group.

**C++ Syntax**

```cpp
FMOD_RESULT SoundGroup::getNumPlaying(
    int * numplaying
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_SoundGroup_GetNumPlaying(
    FMOD_SOUNDBOX * soundgroup,
    int * numplaying
);
```

**Parameters**

`numplaying`

Address of a variable to receive the number of actively playing channels from sounds in this sound group.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This routine returns the number of channels playing. If the sound group only has 1 sound, and that sound is playing twice, the figure returned will be 2.
See Also

- System::createSoundGroup
- System::getMasterSoundGroup
Firelight Technologies FMOD Ex
**SoundGroup::getNumSounds**

Retrieves the current number of sounds in this sound group.

**C++ Syntax**

```cpp
FMOD_RESULT SoundGroup::getNumSounds(
    int * numsounds
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_SoundGroup_GetNumSounds(
    FMOD_SOUNDGROUP * soundgroup,
    int * numsounds
);
```

**Parameters**

`numsounds`

Address of a variable to receive the number of sounds in this sound group.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- `System::createSoundGroup`
- `System::getMasterSoundGroup`
- `SoundGroup::setMaxAudible`
- `SoundGroup::getSound`
Firelight Technologies FMOD Ex
SoundGroup::getSound

Retrieves a pointer to a sound from within a sound group.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getSound(
    int index,
    FMOD::Sound ** sound
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetSound(
    FMOD_SOUNDGROUP * soundgroup,
    int index,
    FMOD_SOUND ** sound
);
```

Parameters

`index`

Index of the sound that is to be retrieved.

`sound`

Address of a variable to receive a pointer to a Sound object.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks
Use `SoundGroup::getNumSounds` in conjunction with this function to enumerate all sounds in a sound group.

**See Also**

- `System::createSoundGroup`
- `System::createSound`
- `SoundGroup::getNumSounds`
- `System::getMasterSoundGroup`
Firelight Technologies FMOD Ex
SoundGroup::getSystemObject

Retrieves the parent System object that was used to create this object.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getSystemObject(
    FMOD::System ** system
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetSystemObject(
    FMOD_SOUNDGROUP * soundgroup,
    FMOD_SYSTEM ** system
);
```

Parameters

**system**

Address of a pointer that receives the System object.

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- System::createSoundGroup
- System::getMasterSoundGroup
Firelight Technologies FMOD Ex
SoundGroup::getUserData

Retrieves the user value that that was set by calling the SoundGroup::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetUserData(
    FMOD_SOUNDGROUP * soundgroup,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the SoundGroup::setUserData function.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- SoundGroup::setUserData
- System::createSoundGroup
• **System::getMasterSoundGroup**
Firelight Technologies FMOD Ex
SoundGroup::getVolume

Retrieves the volume for the sounds within a soundgroup.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::getVolume(
    float * volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_GetVolume(
    FMOD_SOUNDGROUP * soundgroup,
    float * volume
);
```

Parameters

`volume`

Address of a variable to receive the soundgroup volume level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- SoundGroup::setVolume
- System::createSoundGroup
- System::getMasterSoundGroup
Firelight Technologies FMOD Ex
SoundGroup::release

Releases a soundgroup object and returns all sounds back to the master sound group.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::release();
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_Release(FMOD_SOUNDGROUP * soundgroup);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

You cannot release the master sound group.

See Also

- [System::createSoundGroup](#)
- [System::getMasterSoundGroup](#)
Firelight Technologies FMOD Ex
SoundGroup::setMaxAudible

Limits the number of concurrent playbacks of sounds in a sound group to the specified value.
After this, if the sounds in the sound group are playing this many times, any attempts to play more of the sounds in the sound group will by default fail with FMOD_ERR_MAXAUDIBLE.
Use SoundGroup::setMaxAudibleBehavior to change the way the sound playback behaves when too many sounds are playing. Muting, failing and stealing behaviors can be specified.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::setMaxAudible(
    int maxaudible
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_SetMaxAudible(
    FMOD_SOUNDDATA * soundgroup,
    int maxaudible
);
```

Parameters

`maxaudible`

Number of playbacks to be audible at once. -1 = unlimited. 0 means no sounds in this group will succeed. Default = -1.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

`SoundGroup::getNumPlaying` can be used to determine how many instances of the sounds in the sound group are currently playing.

See Also

- `System::createSoundGroup`
- `SoundGroup::getMaxAudible`
- `SoundGroup::getNumPlaying`
- `SoundGroup::setMaxAudibleBehavior`
- `SoundGroup::getMaxAudibleBehavior`
- `System::getMasterSoundGroup`
Firelight Technologies FMOD Ex
SoundGroup::setMaxAudibleBehavior

This function changes the way the sound playback behaves when too many sounds are playing in a soundgroup. Muting, failing and stealing behaviors can be specified.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::setMaxAudibleBehavior(  
   FMOD_SOUNDGROUP_BEHAVIOR behavior
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_SetMaxAudibleBehavior(  
   FMOD_SOUNDGROUP * soundgroup,  
   FMOD_SOUNDGROUP_BEHAVIOR behavior
);
```

Parameters

behavior

Specify a behavior determined with a FMOD_SOUNDGROUP_BEHAVIOR flag. Default is FMOD_SOUNDGROUP_BEHAVIOR_FAIL.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also
- **FMOD_SOUNDCOUNT_BEHAVIOR**
- **SoundGroup::getMaxAudibleBehavior**
- **SoundGroup::setMaxAudible**
- **SoundGroup::getMaxAudible**
- **SoundGroup::setMuteFadeSpeed**
- **SoundGroup::getMuteFadeSpeed**
- **System::createSoundGroup**
- **System::getMasterSoundGroup**
Firelight Technologies FMOD Ex
SoundGroup::setMuteFadeSpeed

Specify a time in seconds for **FMOD_SOUNDGROUP_BEHAVIOR_MUTE** behavior to fade with. By default there is no fade.

When more sounds are playing in a SoundGroup than are specified with **SoundGroup::setMaxAudible**, the least important sound (ie lowest priority / lowest audible volume due to 3d position, volume etc) will fade to silence if **FMOD_SOUNDGROUP_BEHAVIOR_MUTE** is used, and any previous sounds that were silent because of this rule will fade in if they are more important.

**C++ Syntax**

```cpp
FMOD_RESULT SoundGroup::setMuteFadeSpeed(
    float speed
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_SoundGroup_SetMuteFadeSpeed(
    FMOD_SOUNDGROUP * soundgroup,
    float speed
);
```

**Parameters**

* speed

Fade time in seconds (1.0 = 1 second). Default = 0.0. (no fade).

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

If a mode besides `FMOD_SOUNDFGROUP_BEHAVIOR_MUTE` is used, the fade speed is ignored.

See Also

- `SoundGroup::getMuteFadeSpeed`
- `SoundGroup::setMaxAudibleBehavior`
- `SoundGroup::getMaxAudibleBehavior`
- `SoundGroup::setMaxAudible`
- `SoundGroup::getMaxAudible`
- `System::createSoundGroup`
- `System::getMasterSoundGroup`
Firelight Technologies FMOD Ex
SoundGroup::setUserData

Sets a user value that the SoundGroup object will store internally. Can be retrieved with SoundGroup::getUserData.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_SetUserData(
    FMOD_SOUNDGROUP * soundgroup,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the sound group object.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using \texttt{SoundGroup::getUserData} would help in the identification of the object.

\section*{See Also}

- \texttt{SoundGroup::getUserData}
- \texttt{System::createSoundGroup}
- \texttt{System::getMasterSoundGroup}
Firelight Technologies FMOD Ex
SoundGroup::setVolume

Sets the volume for a sound group, affecting all channels playing the sounds in this soundgroup.

C++ Syntax

```cpp
FMOD_RESULT SoundGroup::setVolume(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_SoundGroup_SetVolume(
    FMOD_SOUNDGROUP * soundgroup,
    float volume
);
```

Parameters

`volume`

A linear volume level, from 0.0 to 1.0 inclusive. 0.0 = silent, 1.0 = full volume. Default = 1.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- SoundGroup::getVolume
- System::createSoundGroup
• `System::getMasterSoundGroup`
Firelight Technologies FMOD Ex
# SoundGroup::stop

Stops all sounds within this soundgroup.

**C++ Syntax**

```cpp
FMOD_RESULT SoundGroup::stop();
```

**C Syntax**

```c
FMOD_RESULT FMOD_SoundGroup_Stop(FMOD_SOUNDGROUP * soundgroup);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [System::playSound](#)
- [System::createSoundGroup](#)
- [System::getMasterSoundGroup](#)
Firelight Technologies FMOD Ex
DSP Interface

DSP::addInput  DSP::disconnectAll
DSP::disconnectFrom
DSP::getActive
DSP::getBypass
DSP::getDefaults
DSP::getInfo
DSP::getInput
DSP::getMemoryInfo
DSP::getNumInputs
DSP::getNumOutputs
DSP::getNumParameters
DSP::getOutput
DSP::getParameter
DSP::getParameterInfo
DSP::getSpeakerActive
DSP::getSystemObject
DSP::getType
DSP::getUserData
DSP::release
DSP::remove
DSP::reset
DSP::setActive
DSP::setBypass
DSP::setDefaults
DSP::setParameter
DSP::setSpeakerActive
DSP::setUserData
DSP::showConfigDialog
Firelight Technologies FMOD Ex
DSP::addInput

Adds the specified DSP unit as an input of the DSP object.

C++ Syntax

```cpp
FMOD_RESULT DSP::addInput(
    FMOD::DSP * target,
    FMOD::DSPConnection ** connection
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_AddInput(
    FMOD_DSP * dsp,
    FMOD_DSP * target,
    FMOD_DSPCONNECTION ** connection
);
```

Parameters

`target`

The DSP unit to add as an input of the current unit.

`connection`

The connection between the 2 units. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Adding a unit as an input means that there can be multiple units added to the target.

Inputs are automatically mixed together, then the mixed data is sent to the unit's
output(s).

To find the number of inputs or outputs a unit has use `DSP::getNumInputs` or `DSP::getNumOutputs`.

**Note:** The connection pointer retrieved here will become invalid if you disconnect the 2 dsp units that use it.

**Remarks**

If you want to add a unit as an output of another unit, then add 'this' unit as an input of that unit instead.

**See Also**

- `DSP::getNumInputs`
- `DSP::getInput`
- `DSP::getNumOutputs`
- `DSP::disconnectFrom`
Firelight Technologies FMOD Ex
DSP::disconnectAll

Helper function to disconnect either all inputs or all outputs of a dsp unit.

C++ Syntax

```cpp
FMOD_RESULT DSP::disconnectAll(
    bool inputs,
    bool outputs
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_DisconnectAll(
    FMOD_DSP * dsp,
    FMOD_BOOL inputs,
    FMOD_BOOL outputs
);
```

Parameters

*inputs*

true = disconnect all inputs to this DSP unit. false = leave input connections alone.

*outputs*

true = disconnect all outputs to this DSP unit. false = leave output connections alone.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

This function is optimized to be faster than disconnecting inputs and outputs manually one by one.

**Important note:** If you have a handle to DSPConnection pointers that bind any of the inputs or outputs to this DSP unit, then they will become invalid. The connections are sent back to a freelist to be re-used again by a later addInput command.

See Also

- [DSP::disconnectFrom](#)
Firelight Technologies FMOD Ex
DSP::disconnectFrom

Disconnect the DSP unit from the specified target.

C++ Syntax

```cpp
FMOD_RESULT DSP::disconnectFrom(
    FMOD::DSP * target
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_DisconnectFrom(
    FMOD_DSP * dsp,
    FMOD_DSP * target
);
```

Parameters

target

The unit that this unit is to be removed from. Specify 0 or NULL to disconnect the unit from all outputs and inputs.

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

Note that when you disconnect a unit, it is up to you to reconnect the network so that data flow can continue.
**Important note:** If you have a handle to the connection pointer that binds these 2 DSP units, then it will become invalid. The connection is then sent back to a freelist to be re-used again by a later addInput command.

**See Also**

- [DSP::addInput](#)
- [DSP::disconnectAll](#)
Firelight Technologies FMOD Ex
**DSP::getActive**

Retrieves the active state of a DSP unit.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::getActive(
    bool * active
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_GetActive(
    FMOD_DSP * dsp,
    FMOD_BOOL * active
);
```

**Parameters**

*active*

Address of a variable that receives the active state of the unit. true = unit is activated, false = unit is deactivated.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [DSP::setActive](#)
- [DSP::setBypass](#)
Firelight Technologies FMOD Ex
DSP::getBypass

Retrieves the bypass state of the DSP unit.

C++ Syntax

```cpp
FMOD_RESULT DSP::getBypass(
    bool * bypass
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetBypass(
    FMOD_DSP * dsp,
    FMOD_BOOL * bypass
);
```

Parameters

*bypass*

Address of a variable that receives the bypass state for a DSP unit. true = unit is not processing audio data, false = unit is processing audio data. Default = false.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

If a unit is bypassed, it will still process its inputs, unlike `DSP::setActive` (when set to false) which causes inputs to stop processing as well.
See Also

- DSP::setBypass
- DSP::setActive
Firelight Technologies FMOD Ex
DSP::getDefaults

Retrieves the default frequency, volume, pan and more for this DSP unit if it was to ever be played on a channel using System::playDSP.

C++ Syntax

```cpp
FMOD_RESULT DSP::getDefaults(
    float * frequency,
    float * volume,
    float * pan,
    int * priority
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetDefaults(
    FMOD_DSP * dsp,
    float * frequency,
    float * volume,
    float * pan,
    int * priority
);
```

Parameters

**frequency**

Address of a variable that receives the default frequency for the DSP unit. Optional. Specify 0 or NULL to ignore.

**volume**

Address of a variable that receives the default volume for the DSP unit. Result will be from 0.0 to 1.0. 0.0 = Silent, 1.0 = full volume. Default = 1.0. Optional. Specify 0 or NULL to ignore.

**pan**

Address of a variable that receives the default pan for the DSP unit. Result will
be from -1.0 to +1.0. -1.0 = Full left, 0.0 = center, 1.0 = full right. Default = 0.0. Optional. Specify 0 or NULL to ignore.

*priority*

Address of a variable that receives the default priority for the DSP unit when played on a channel. Result will be from 0 to 256. 0 = most important, 256 = least important. Default = 128. Optional. Specify 0 or NULL to ignore.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- DSP::setDefaults
- System::playDSP
Firelight Technologies FMOD Ex
DSP::getInfo

Retrieves information about the current DSP unit, including name, version, default channels and width and height of configuration dialog box if it exists.

C++ Syntax

```cpp
FMOD_RESULT DSP::getInfo(
    char * name,
    unsigned int * version,
    int * channels,
    int * configwidth,
    int * configheight
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetInfo(
    FMOD_DSP * dsp,
    char * name,
    unsigned int * version,
    int * channels,
    int * configwidth,
    int * configheight
);
```

Parameters

**name**

Address of a variable that receives the name of the unit. This will be a maximum of 32 bytes. If the DSP unit has filled all 32 bytes with the name with no terminating \0 null character it is up to the caller to append a null character. Optional. Specify 0 or NULL to ignore.

**version**

Address of a variable that receives the version number of the DSP unit. Version number is usually formatted as hex AAAABBBB where the AAAA is the major version number and the B BBB is the minor version number. Optional. Specify 0 or NULL to ignore.
channels

Address of a variable that receives the number of channels the unit was initialized with. 0 means the plugin will process whatever number of channels is currently in the network. >0 would be mostly used if the unit is a unit that only generates sound, or is not flexible enough to take any number of input channels. Optional. Specify 0 or NULL to ignore.

configwidth

Address of a variable that receives the width of an optional configuration dialog box that can be displayed with DSP::showConfigDialog. 0 means the dialog is not present. Optional. Specify 0 or NULL to ignore.

configheight

Address of a variable that receives the height of an optional configuration dialog box that can be displayed with DSP::showConfigDialog. 0 means the dialog is not present. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- DSP::showConfigDialog
Firelight Technologies FMOD Ex
DSP::getInput

Retrieves a pointer to a DSP unit which is acting as an input to this unit.

C++ Syntax

```cpp
FMOD_RESULT DSP::getInput(
    int index,
    FMOD::DSP ** input,
    FMOD::DSPConnection ** inputconnection
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetInput(
    FMOD_DSP * dsp,
    int index,
    FMOD_DSP ** input,
    FMOD_DSPCONNECTION ** inputconnection
);
```

Parameters

`index`

Index of the input unit to retrieve.

`input`

Address of a variable that receives the pointer to the desired input unit.

`inputconnection`

The connection between the 2 units. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is `FMOD_OK`. 
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

An input is a unit which feeds audio data to this unit. If there are more than 1 input to this unit, the inputs will be mixed, and the current unit processes the mixed result. Find out the number of input units to this unit by calling `DSP::getNumInputs`.

**Performance warning!** Because this function needs to flush the dsp queue before it can determine if the specified numerical input is available or not, this function may block significantly while the background mixer thread operates.

**Note:** The connection pointer retrieved here will become invalid if you disconnect the 2 dsp units that use it.

**See Also**

- `DSP::getNumInputs`
- `DSP::addInput`
- `DSP::getOutput`
- `DSPConnection::getMix`
- `DSPConnection::setMix`
Firelight Technologies FMOD Ex
DSP::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT DSP::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetMemoryInfo(
    FMOD_DSP * dsp,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

`memorybits`

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

`event_memorybits`

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

`memoryused`

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

`memoryused_details`
Optional. Specify 0 to ignore. Address of a user-allocated `FMOD_MEMORY_USAGE_DETAILS` structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See `System::getMemoryInfo` for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGE_DETAILS`
- `System::getMemoryInfo`
Firelight Technologies FMOD Ex
DSP::getNumInputs

Retrieves the number of inputs connected to the DSP unit.

C++ Syntax

```cpp
FMOD_RESULT DSP::getNumInputs(
    int * numinputs
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetNumInputs(
    FMOD_DSP * dsp,
    int * numinputs
);
```

Parameters

`numinputs`

Address of a variable that receives the number of inputs connected to this unit.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Inputs are units that feed data to this unit. When there are multiple inputs, they are mixed together.

Performance warning! Because this function needs to flush the dsp queue
before it can determine how many units are available, this function may block significantly while the background mixer thread operates.

See Also

- DSP::getNumOutputs
- DSP::getInput
Firelight Technologies FMOD Ex
DSP::getNumOutputs

Retrieves the number of outputs connected to the DSP unit.

C++ Syntax

```cpp
FMOD_RESULT DSP::getNumOutputs(
    int * numoutputs
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetNumOutputs(
    FMOD_DSP * dsp,
    int * numoutputs
);
```

Parameters

`numoutputs`

Address of a variable that receives the number of outputs connected to this unit.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Outputs are units that this unit feeds data to. When there are multiple outputs, the data is split and sent to each unit individually.

**Performance warning!** Because this function needs to flush the dsp queue
before it can determine how many units are available, this function may block significantly while the background mixer thread operates.

See Also

- `DSP::getNumInputs`
- `DSP::getOutput`
Firelight Technologies FMOD Ex
DSP::getNumParameters

Retrieves the number of parameters a DSP unit has to control its behaviour.

C++ Syntax

```cpp
FMOD_RESULT DSP::getNumParameters(
    int * numparams
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetNumParameters(
    FMOD_DSP * dsp,
    int * numparams
);
```

Parameters

`numparams`

Address of a variable that receives the number of parameters contained within this DSP unit.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Use this to enumerate all parameters of a DSP unit with `DSP::getParameter` and `DSP::getParameterInfo`. 
See Also

- DSP::setParameter
- DSP::getParameter
- DSP::getParameterInfo
Firelight Technologies FMOD Ex
DSP::getOutput

Retrieves a pointer to a DSP unit which is acting as an output to this unit.

C++ Syntax

```cpp
FMOD_RESULT DSP::getOutput(
    int index,
    FMOD::DSP ** output,
    FMOD::DSPConnection ** outputconnection
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetOutput(
    FMOD_DSP * dsp,
    int index,
    FMOD_DSP ** output,
    FMOD_DSPCONNECTION ** outputconnection
);
```

Parameters

index

Index of the output unit to retrieve.

output

Address of a variable that receives the pointer to the desired output unit.

outputconnection

The connection between the 2 units. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

An output is a unit which this unit will feed data too once it has processed its data. Find out the number of output units to this unit by calling `DSP::getNumOutputs`.

**Performance warning!** Because this function needs to flush the dsp queue before it can determine if the specified numerical output is available or not, this function may block significantly while the background mixer thread operates.

**Note:** The connection pointer retrieved here will become invalid if you disconnect the 2 dsp units that use it.

**See Also**

- `DSP::getNumOutputs`
- `DSP::addInput`
- `DSP::getInput`
- `DSPConnection::getMix`
- `DSPConnection::setMix`
Firelight Technologies FMOD Ex
DSP::getParameter

Retrieves a DSP unit's parameter by index. To find out the parameter names and range, see the see also field.

C++ Syntax

```cpp
FMOD_RESULT DSP::getParameter(
    int index,
    float * value,
    char * valuestr,
    int valuestrlen
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetParameter(
    FMOD_DSP * dsp,
    int index,
    float * value,
    char * valuestr,
    int valuestrlen
);
```

Parameters

index

Parameter index for this unit. Find the number of parameters with DSP::getNumParameters.

value

Address of a variable that receives the parameter value. The parameter properties can be retrieved with DSP::getParameterInfo.

valuestr

Address of a variable that receives the string containing a formatted or more meaningful representation of the DSP parameter's value. For example if a switch
parameter has on and off (0.0 or 1.0) it will display "ON" or "OFF" by using this parameter.

valuestrlen

Length of the user supplied memory in bytes that valuestr will write to. This will not exceed 16 bytes.

Return Values

If the function succeeds then the return value is \texttt{FMOD\_OK}.
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

See Also

- \texttt{DSP::getParameterInfo}
- \texttt{DSP::getNumParameters}
- \texttt{DSP::setParameter}
- \texttt{FMOD\_DSP\_OSCILLATOR}
- \texttt{FMOD\_DSP\_LOWPASS}
- \texttt{FMOD\_DSP\_ITLOWPASS}
- \texttt{FMOD\_DSP\_HIGHPASS}
- \texttt{FMOD\_DSP\_ECHO}
- \texttt{FMOD\_DSP\_DELAY}
- \texttt{FMOD\_DSP\_FLANGE}
- \texttt{FMOD\_DSP\_TREMOLO}
- \texttt{FMOD\_DSP\_DISTORTION}
- \texttt{FMOD\_DSP\_NORMALIZE}
- \texttt{FMOD\_DSP\_PARAMEQ}
- \texttt{FMOD\_DSP\_PITCHSHIFT}
- \texttt{FMOD\_DSP\_CHORUS}
- \texttt{FMOD\_DSP\_ITECHO}
- \texttt{FMOD\_DSP\_COMPRESSOR}
- \texttt{FMOD\_DSP\_SFXREVERB}
- \texttt{FMOD\_DSP\_LOWPASS\_SIMPLE}
• **FMOD_DSP_HIGHPASS_SIMPLE**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
DSP::getParameterInfo

Retrieve information about a specified parameter within the DSP unit.

C++ Syntax

```cpp
FMOD_RESULT DSP::getParameterInfo(
    int index,
    char * name,
    char * label,
    char * description,
    int descriptionlen,
    float * min,
    float * max
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetParameterInfo(
    FMOD_DSP * dsp,
    int index,
    char * name,
    char * label,
    char * description,
    int descriptionlen,
    float * min,
    float * max
);
```

Parameters

**index**

Parameter index for this unit. Find the number of parameters with **DSP::getNumParameters**.

**name**

Address of a variable that receives the name of the parameter. An example is "Gain". This is a maximum string length of 16 bytes (append \0 in case the plugin has used all 16 bytes for the string).
label

Address of a variable that receives the label of the parameter (ie a parameter type that might go next to the parameter). An example is "dB". This is a maximum string length of 16 bytes (append \0 in case the plugin has used all 16 bytes for the string).

description

Address of a variable that receives the more descriptive text about the parameter (ie for a tooltip). An example is "Controls the input level for the effect in decibels".

descriptionlen

Maximum length of user supplied description string in bytes that FMOD will write to.

min

Minimum range of the parameter.

max

Maximum range of the parameter.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Use DSP::getNumParameters to find out the number of parameters for this DSP unit.
See Also

- DSP::setParameter
- DSP::getParameter
- DSP::getNumParameters
Firelight Technologies FMOD Ex
DSP::getSpeakerActive

Retrieves the active state of the effect on the given speaker.

C++ Syntax

```cpp
FMOD_RESULT DSP::getSpeakerActive(
    FMOD_SPEAKER speaker,
    bool * active
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetSpeakerActive(
    FMOD_DSP * dsp,
    FMOD_SPEAKER speaker,
    FMOD_BOOL * active
);
```

Parameters

`speaker`

The speaker channel that is being checked

`active`

Address of a variable that receieves the active state for a DSP unit. true = DSP unit is active on that speakerchannel, false = DSP unit is not active on that speaker channel. Default = true.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- DSP::setSpeakerActive
- DSP::getActive
Firelight Technologies FMOD Ex
DSP::getSystemObject

Retrieves the parent System object that was used to create this object.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::getSystemObject(
    FMOD::System ** system
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_GetSystemObject(
    FMOD_DSP * dsp,
    FMOD_SYSTEM ** system
);
```

**Parameters**

- `system`

Address of a variable that receives the System object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- System::createDSP
- System::createDSPByType
- System::getDSPHead
- Channel::getDSPHead
• ChannelGroup::getDSPHead
Firelight Technologies FMOD Ex
DSP::getType

Retrieves the pre-defined type of a FMOD registered DSP unit.

C++ Syntax

```cpp
FMOD_RESULT DSP::getType(
    FMOD_DSP_TYPE * type
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetType(
    FMOD_DSP * dsp,
    FMOD_DSP_TYPE * type
);
```

Parameters

`type`

Address of a variable to recieve the FMOD dsp type.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This is only valid for built in FMOD effects. Any user plugins will simply return `FMOD_DSP_TYPE_UNKNOWN`. 
See Also

- **FMOD_DSP_TYPE**
Firelight Technologies FMOD Ex
**DSP::getUserData**

Retrieves the user value that that was set by calling the **DSP::setUserData** function.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::getUserData(
    void ** userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_GetUserData(
    FMOD_DSP * dsp,
    void ** userdata
);
```

**Parameters**

`userdata`

Address of a pointer that receives the user data specified with the **DSP::setUserData** function.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- **DSP::setUserData**
Firelight Technologies FMOD Ex
DSP::release

Frees a DSP object.

C++ Syntax

FMOD_RESULT DSP::release();

C Syntax

FMOD_RESULT FMOD_DSP_Release(FMOD_DSP * dsp);

Parameters

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This will free the DSP object.

See Also

- System::createDSP
- System::createDSPByType
- System::getDSPHead
- Channel::getDSPHead
- ChannelGroup::getDSPHead
Firelight Technologies FMOD Ex
DSP::remove

Removes a unit from a DSP chain and connects the unit's input and output together after it is gone.

C++ Syntax

```cpp
FMOD_RESULT DSP::remove();
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_Remove(FMOD_DSP * dsp);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is generally only used with units that have been added with `System::addDSP` or `Channel::addDSP`.
A unit that has been added in this way generally only has one input and one output, so this function assumes this and takes input 0 and connects it with output 0 after it has been removed, so that the data flow is not broken.

**Important note:** If you have a handle to DSPConnection pointers that bind any of the inputs or outputs to this DSP unit, then they will become invalid. The connections are sent back to a freelist to be re-used again by a later addInput command.
**Note:** If the unit has not been added with addDSP it will not restore links, and will just disconnect all inputs and outputs, making it equivalent to disconnectAll(true, true).

**See Also**

- [System::addDSP](#)
- [Channel::addDSP](#)
- [ChannelGroup::addDSP](#)
Firelight Technologies FMOD Ex
DSP::reset

Calls the DSP unit's reset function, which will clear internal buffers and reset the unit back to an initial state.

C++ Syntax

```cpp
FMOD_RESULT DSP::reset();
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_Reset(FMOD_DSP * dsp);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Calling this function is useful if the DSP unit relies on a history to process itself (i.e., an echo filter). If you disconnected the unit and reconnected it to a different part of the network with a different sound, you would want to call this to reset the units state (i.e., clear and reset the echo filter) so that you don't get left over artifacts from the place it used to be connected.
Firelight Technologies FMOD Ex
DSP::setActive

Enables or disables a unit for being processed.

C++ Syntax

```cpp
FMOD_RESULT DSP::setActive(
    bool active
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_SetActive(
    FMOD_DSP * dsp,
    FMOD_BOOL active
);
```

Parameters

active

ture = unit is activated, false = unit is deactivated.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This does not connect or disconnect a unit in any way, it just disables it so that it is not processed.
If a unit is disabled, and has inputs, they will also cease to be processed.
To disable a unit but allow the inputs of the unit to continue being processed, use
DSP::setBypass instead.

See Also

- DSP::getActive
- DSP::setBypass
Firelight Technologies FMOD Ex
**DSP::setBypass**

Enables or disables the read callback of a DSP unit so that it does or doesn't process the data coming into it.

A DSP unit that is disabled still processes its inputs, it will just be 'dry'.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::setBypass(
    bool bypass
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_SetBypass(
    FMOD_DSP * dsp,
    FMOD_BOOL bypass
);
```

**Parameters**

*bypass*

Boolean to cause the read callback of the DSP unit to be bypassed or not. Default = false.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

If a unit is bypassed, it will still process its inputs.
To disable the unit and all of its inputs, use `DSP::setActive` instead.

See Also

- `DSP::getBypass`
- `DSP::setActive`
Firelight Technologies FMOD Ex
DSP::setDefaults

If a DSP unit is to be played on a channel with System::playDSP, this will set the defaults for frequency, volume, pan and more for the channel.

C++ Syntax

```cpp
FMOD_RESULT DSP::setDefaults(
    float frequency,
    float volume,
    float pan,
    int priority
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_SetDefaults(
    FMOD_DSP * dsp,
    float frequency,
    float volume,
    float pan,
    int priority
);
```

Parameters

*frequency*

Default playback frequency for the DSP unit, in hz. (ie 44100hz).

*volume*

Default volume for the DSP unit. 0.0 to 1.0. 0.0 = Silent, 1.0 = full volume. Default = 1.0.

*pan*

Default pan for the DSP unit. -1.0 to +1.0. -1.0 = Full left, 0.0 = center, 1.0 = full right. Default = 0.0.

*priority*
Default priority for the DSP unit when played on a channel. 0 to 256. 0 = most important, 256 = least important. Default = 128.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

There are no 'ignore' values for these parameters. Use **DSP::getDefaults** if you want to change only 1 and leave others unaltered.

**See Also**

- **System::playDSP**
- **DSP::getDefaults**
Firelight Technologies FMOD Ex
DSP::setParameter

Sets a DSP unit's parameter by index. To find out the parameter names and range, see the see also field.

C++ Syntax

```cpp
FMOD_RESULT DSP::setParameter(
    int index,
    float value
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_SetParameter(
    FMOD_DSP * dsp,
    int index,
    float value
);
```

Parameters

`index`

Parameter index for this unit. Find the number of parameters with **DSP::getNumParameters**.

`value`

Parameter value. The parameter properties can be retrieved with **DSP::getParameterInfo**.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
See Also

- DSP::getParameterInfo
- DSP::getNumParameters
- DSP::getParameter
- FMOD_DSP_OSCILLATOR
- FMOD_DSP_LOWPASS
- FMOD_DSP_ITLOWPASS
- FMOD_DSP_HIGHPASS
- FMOD_DSP_ECHO
- FMOD_DSP_DELAY
- FMOD_DSP_FLANGE
- FMOD_DSP_TREMOLO
- FMOD_DSP_DISTORTION
- FMOD_DSP_NORMALIZE
- FMOD_DSP_PARAMEQ
- FMOD_DSP_PITCHSHIFT
- FMOD_DSP_CHORUS
- FMOD_DSP_ITECHO
- FMOD_DSP_COMPRESSOR
- FMOD_DSP_SFXREVERB
- FMOD_DSP_LOWPASS_SIMPLE
- FMOD_DSP_HIGHPASS_SIMPLE
Firelight Technologies FMOD Ex
DSP::setSpeakerActive

Enables or disables the DSP effect on the given speaker channel. This is used to reduce the overhead of DSP effect by only applying the effect to the speaker channels where it is needed.

C++ Syntax

```cpp
FMOD_RESULT DSP::setSpeakerActive(
    FMOD_SPEAKER speaker,
    bool active
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_SetSpeakerActive(
    FMOD_DSP * dsp,
    FMOD_SPEAKER speaker,
    FMOD_BOOL active
);
```

Parameters

`speaker`

The speaker channel that is being set.

`active`

Boolean to cause the DSP to be active/inactive on a given speaker channel. Default = true.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

If a speaker channel is deactivated it will not have the DSP effect applied to it.

See Also

- DSP::getSpeakerActive
- DSP::setActive
Firelight Technologies FMOD Ex
DSP::setUserData

Sets a user value that the DSP object will store internally. Can be retrieved with DSP::getUserData.

C++ Syntax

```cpp
FMOD_RESULT DSP::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_SetUserData(
    FMOD_DSP * dsp,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the DSP object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object. It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using **DSP::getUserData** would help in the identification of the object.

**See Also**

- **DSP::getUserData**
Firelight Technologies FMOD Ex
DSP::showConfigDialog

Display or hide a DSP unit configuration dialog box inside the target window.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::showConfigDialog(
    void * hwnd,
    bool  show
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_ShowConfigDialog(
    FMOD_DSP * dsp,
    void * hwnd,
    FMOD_BOOL  show
);
```

**Parameters**

`hwnd`

Target HWND in windows to display configuration dialog.

`show`

true = show dialog box inside target hwnd. false = remove dialog from target hwnd.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

Dialog boxes are used by DSP plugins that prefer to use a graphical user interface to modify their parameters rather than using the other method of enumerating the parameters and using `DSP::setParameter`. These are usually VST plugins. FMOD Ex plugins do not have configuration dialog boxes. To find out what size window to create to store the configuration screen, use `DSP::getInfo` where you can get the width and height.

See Also

- `DSP::getInfo`
- `DSP::setParameter`
- `DSP::getParameter`
Firelight Technologies FMOD Ex
DSPConnection Interface

DSPConnection::getInput  DSPConnection::getLevels
DSPConnection::getMemoryInfo
DSPConnection::getMix
DSPConnection::getOutput
DSPConnection::getUserData
DSPConnection::setLevels
DSPConnection::setMix
DSPConnection::setUserData
Firelight Technologies FMOD Ex
DSPConnection::getInput

Retrieves the DSP unit that is the input of this connection.

**C++ Syntax**

```cpp
FMOD_RESULT DSPConnection::getInput(
    FMOD::DSP ** input
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSPConnection_GetInput(
    FMOD_DSPCONNECTION * dspconnection,
    FMOD_DSP ** input
);
```

**Parameters**

`input`

Address of a pointer that receives the pointer to the DSP unit that is the input of this connection.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

A DSPConnection joins 2 DSP units together (think of it as the line between 2 circles).
Each DSPConnection has 1 input and 1 output.
Note! If a DSP::addInput just occurred, the connection might not be ready because the DSP system is still queued to connect in the background. If so the function will return FMOD_ERR_NOTREADY and the input will be null. Poll until it is ready.

See Also

- DSPConnection::getOutput
- DSP::addInput
Firelight Technologies FMOD Ex
**DSPConnection::getLevels**

Retrieves the speaker mix for a DSP connection for a particular output speaker.

**C++ Syntax**

```cpp
FMOD_RESULT DSPConnection::getLevels(
    FMOD_SPEAKER speaker,
    float * levels,
    int numlevels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSPConnection_GetLevels(
    FMOD_DSPCONNECTION * dspconnection,
    FMOD_SPEAKER speaker,
    float * levels,
    int numlevels
);
```

**Parameters**

* speaker

The target speaker to get the levels from. This can be cast to an integer if you are using a device with more than the pre-defined speaker range.

* levels

Address of an array of floating point numbers to get the speaker levels of an input.

* numlevels

The number of floats within the levels parameter being passed to this function. In the case of the above mono or stereo sound, 1 or 2 could be used respectively. If the sound being played was an 8 channel multichannel sound then 8 levels would be used.
**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Note! To conserve memory the stored levels are converted from floating point to 16bit integers (4.12 fixed point). This means when using DSP::getInputLevels the values may not come back exactly as they were set.

**See Also**

- DSPConnection::setLevels
- DSPConnection::getInput
- DSPConnection::getOutput
- DSPConnection::setMix
Firelight Technologies FMOD Ex
DSPConnection::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_GetMemoryInfo(
    FMOD_DSPCONNECTION * dspconnection,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

**memorybits**

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

**event_memorybits**

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

**memoryused**

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

**memoryused_details**
Optional. Specify 0 to ignore. Address of a user-allocated `FMOD_MEMORY_USAGEDETAILS` structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See `System::getMemoryInfo` for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGEDETAILS`
- `System::getMemoryInfo`
Firelight Technologies FMOD Ex
DSPConnection::getMix

Retrieves the volume of the connection - the scale level of the input before being passed to the output.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::getMix(
    float * volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_GetMix(
    FMOD_DSPCONNECTION * dspconnection,
    float * volume
);
```

Parameters

`volume`

Address of a variable to receive the volume or mix level of the specified input. 0.0 = silent, 1.0 = full volume.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- DSPConnection::setMix
- DSP::getInput
- **DSP::getOutput**
Firelight Technologies FMOD Ex
DSPConnection::getOutput

Retrieves the DSP unit that is the output of this connection.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::getOutput( 
    FMOD::DSP ** output
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_GetOutput( 
    FMOD_DSPCONNECTION * dspconnection, 
    FMOD_DSP ** output
);
```

Parameters

`output`

Address of a pointer that receives the pointer to the DSP unit that is the output of this connection.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

A DSPConnection joins 2 DSP units together (think of it as the line between 2 circles).
Each DSPConnection has 1 input and 1 output.
Note! If a **DSP::addInput** just occurred, the connection might not be ready because the DSP system is still queued to connect in the background. If so the function will return **FMOD_ERR_NOTREADY** and the input will be null. Poll until it is ready.

See Also

- **DSPConnection::getInput**
- **DSP::addInput**
Firelight Technologies FMOD Ex
DSPConnection::getUserData

Sets a user value that the DSPConnection object will store internally. Can be retrieved with DSPConnection::getUserData.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_GetUserData(
    FMOD_DSPCONNECTION * dspconnection,
    void ** userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the DSPConnection object.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
See Also

- DSPConnection::getUserData
Firelight Technologies FMOD Ex
DSPConnection::setLevels

For a particular speaker, the levels of the incoming channels of the connection are set so that they will be scaled before being passed to the output.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::setLevels(
    FMOD_SPEAKER speaker,
    float * levels,
    int numlevels
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_SetLevels(
    FMOD_DSPCONNECTION * dspconnection,
    FMOD_SPEAKER speaker,
    float * levels,
    int numlevels
);
```

Parameters

**speaker**

The target speaker to modify the levels for. This can be cast to an integer if you are using a device with more than the pre-defined speaker range.

**levels**

An array of floating point numbers from 0.0 to 1.0 representing the volume of each input channel of a sound. See remarks for more.

**numlevels**

The number of floats within the levels parameter being passed to this function. In the case of the above mono or stereo sound, 1 or 2 could be used respectively. If the sound being played was an 8 channel multichannel sound then 8 levels would be used.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

As an example of usage of this function, if the sound played on this speaker was mono, only 1 level would be needed. If the sound played on this channel was stereo, then an array of 2 floats could be specified. For example `{ 0, 1 }` on a channel playing a stereo sound would mute the left part of the stereo sound when it is played on this speaker.

Note! To conserve memory the levels are converted from floating point to 16bit integers (4.12 fixed point). This means when using `DSPConnection::getLevels` the values may not come back exactly as they were set.

See Also

- `DSPConnection::getLevels`
- `DSPConnection::getInput`
- `DSPConnection::getOutput`
- `DSPConnection::setMix`
Firelight Technologies FMOD Ex
DSPConnection::setMix

Sets the volume of the connection so that the input is scaled by this value before being passed to the output.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::setMix(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_SetMix(
    FMOD_DSPCONNECTION * dspconnection,
    float volume
);
```

Parameters

`volume`

Volume or mix level of the connection. 0.0 = silent, 1.0 = full volume.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [DSPConnection::getMix](#)
- [DSP::getInput](#)
- [DSP::getOutput](#)
Firelight Technologies FMOD Ex
DSPConnection::setUserData

Sets a user value that the DSPConnection object will store internally. Can be retrieved with DSPConnection::getUserData.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_SetUserData(
    FMOD_DSPCONNECTION * dspconnection,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the DSPConnection object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
See Also

- DSPConnection::getUserData
Firelight Technologies FMOD Ex
Geometry Interface

Geometry::addPolygon  Geometry::getActive
Geometry::getMaxPolygons
Geometry::getMemoryInfo
Geometry::getNumPolygons
Geometry::getPolygonAttributes
Geometry::getPolygonNumVertices
Geometry::getPolygonVertex
Geometry::getPosition
Geometry::getRotation
Geometry::getScale
Geometry::getUserData
Geometry::release
Geometry::save
Geometry::setActive
Geometry::setPolygonAttributes
Geometry::setPolygonVertex
Geometry::setPosition
Geometry::setRotation
Geometry::setScale
Geometry::setUserData
Firelight Technologies FMOD Ex
Geometry::addPolygon

Adds a polygon to an existing geometry object.

C++ Syntax

```cpp
FMOD_RESULT Geometry::addPolygon(
    float directocclusion,
    float reverbocclusion,
    bool doublesided,
    int numvertices,
    const FMOD_VECTOR * vertices,
    int * polygonindex
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_AddPolygon(
    FMOD_GEOMETRY * geometry,
    float directocclusion,
    float reverbocclusion,
    FMOD_BOOL doublesided,
    int numvertices,
    const FMOD_VECTOR * vertices,
    int * polygonindex
);
```

Parameters

`directocclusion`

Occlusion value from 0.0 to 1.0 which affects volume or audible frequencies. 0.0 = The polygon does not occlude volume or audible frequencies (sound will be fully audible), 1.0 = The polygon fully occludes (sound will be silent).

`reverbocclusion`

Occlusion value from 0.0 to 1.0 which affects the reverb mix. 0.0 = The polygon does not occlude reverb (reverb reflections still travel through this polygon), 1.0 = The poly fully occludes reverb (reverb reflections will be silent through this polygon).
doublesided

Description of polygon if it is double sided or single sided. true = polygon is double sided, false = polygon is single sided, and the winding of the polygon (which determines the polygon's normal) determines which side of the polygon will cause occlusion.

numvertices

Number of vertices in this polygon. This must be at least 3. Polygons (more than 3 sides) are supported.

vertices

A pointer to an array of vertices located in object space, with the count being the number of vertices described using the numvertices parameter.

polygonindex

Address of a variable to receive the polygon index for this object. This index can be used later with other per polygon based geometry functions.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Note!
- All vertices must lay in the same plane otherwise behaviour may be unpredictable.
- The polygon is assumed to be convex. A non convex polygon will produce unpredictable behaviour.
- Polygons with zero area will be ignored.
Vertices of an object are in object space, not world space, and so are relative to the position, or center of the object. See `Geometry::setPosition`.

See Also

- `Geometry::getNumPolygons`
- `Geometry::setPosition`
- `FMOD_VECTOR`
Firelight Technologies FMOD Ex
Geometry::getActive

Retrieves the user set active state of the geometry object.

**C++ Syntax**

```cpp
FMOD_RESULT Geometry::getActive(
    bool * active
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Geometry_GetActive(
    FMOD_GEOMETRY * geometry,
    FMOD_BOOL * active
);
```

**Parameters**

*active*

Address of a variable to receive the active state of the object. true = active, false = not active. Default = true.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- Geometry::setActive
Firelight Technologies FMOD Ex
Geometry::getMaxPolygons

Retrieves the maximum number of polygons and vertices allocatable for this object. This is not the number of polygons or vertices currently present. The maximum number was set with System::createGeometry.

C++ Syntax

```cpp
FMOD_RESULT Geometry::getMaxPolygons(
    int * maxpolygons,
    int * maxvertices
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetMaxPolygons(
    FMOD_GEOMETRY * geometry,
    int * maxpolygons,
    int * maxvertices
);
```

Parameters

`maxpolygons`

Address of a variable to receive the maximum possible number of polygons in this object.

`maxvertices`

Address of a variable to receive the maximum possible number of vertices in this object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- System::createGeometry
- System::loadGeometry
Firelight Technologies FMOD Ex
Geometry::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT Geometry::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetMemoryInfo(
    FMOD_GEOMETRY * geometry,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

event_memorybits

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

memoryused

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated `FMOD_MEMORY_USAGE_DETAILS` structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See [System::getMemoryInfo](#) for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGE_DETAILS`
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
Geometry::getNumPolygons

Retrieves the number of polygons stored within this geometry object.

C++ Syntax

```cpp
FMOD_RESULT Geometry::getNumPolygons(
    int * numpolygons
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetNumPolygons(
    FMOD_GEOMETRY * geometry,
    int * numpolygons
);
```

Parameters

*numpolygons*

Address of a variable to receive the number of polygons within this object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Polygons are added to a geometry object via Geometry::addPolygon.

See Also
- **Geometry::AddPolygon**
Geometry::getPolygonAttributes

Retrieves the attributes for a particular polygon inside a geometry object.

C++ Syntax

```cpp
FMOD_RESULT Geometry::getPolygonAttributes(
    int index,
    float * directocclusion,
    float * reverbocclusion,
    bool * doublesided
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetPolygonAttributes(
    FMOD_Geometry * geometry,
    int index,
    float * directocclusion,
    float * reverbocclusion,
    FMOD_BOOL * doublesided
);
```

Parameters

index

Polygon index inside the object.

directocclusion

Address of a variable to receive the occlusion value from 0.0 to 1.0 which affects volume or audible frequencies. 0.0 = The polygon does not occlude volume or audible frequencies (sound will be fully audible), 1.0 = The polygon fully occludes (sound will be silent).

reverbocclusion

Address of a variable to receive the occlusion value from 0.0 to 1.0 which affects the reverb mix. 0.0 = The polygon does not occlude reverb (reverb


reflections still travel through this polygon), 1.0 = The polyfully fully occludes reverb (reverb reflections will be silent through this polygon).

*doublesided*

Address of a variable to receieve the description of polygon if it is double sided or single sided. true = polygon is double sided, false = polygon is single sided, and the winding of the polygon (which determines the polygon's normal) determines which side of the polygon will cause occlusion.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [Geometry::getPolygonAttributes](#)
- [Geometry::getNumPolygons](#)
Firelight Technologies FMOD Ex
Geometry::getPolygonNumVertices

Gets the number of vertices in a polygon which is part of the geometry object.

**C++ Syntax**

```cpp
FMOD_RESULT Geometry::getPolygonNumVertices(
    int index,
    int * numvertices
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Geometry_GetPolygonNumVertices(
    FMOD_GEOMETRY * geometry,
    int index,
    int * numvertices
);
```

**Parameters**

*index*

Polygon index. This must be in the range of 0 to Geometry::getNumPolygons minus 1.

*numvertices*

Address of a variable to receive the number of vertices for the selected polygon.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
See Also

- `Geometry::getNumPolygons`
Firelight Technologies FMOD Ex
Geometry::getPolygonVertex

Retrieves the position of the vertex inside a geometry object.

C++ Syntax

```cpp
FMOD_RESULT Geometry::getPolygonVertex(
    int index,
    int vertexindex,
    FMOD_VECTOR * vertex
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetPolygonVertex(
    FMOD_GEOMETRY * geometry,
    int index,
    int vertexindex,
    FMOD_VECTOR * vertex
);
```

Parameters

`index`

Polygon index. This must be in the range of 0 to Geometry::getNumPolygons minus 1.

`vertexindex`

Vertex index inside the polygon. This must be in the range of 0 to Geometry::getPolygonNumVertices minus 1.

`vertex`

Address of an FMOD_VECTOR structure which will receive the new vertex location in object space.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

Vertices are relative to the position of the object. See **Geometry::setPosition**.

See Also

- **Geometry::getPolygonNumVertices**
- **Geometry::setPosition**
- **Geometry::getNumPolygons**
- **FMOD_VECTOR**
Geometry::getPosition

Retrieves the position of the object in 3D world space.

C++ Syntax

```cpp
FMOD_RESULT Geometry::getPosition(
    FMOD_VECTOR * position
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetPosition(
    FMOD_GEOMETRY * geometry,
    FMOD_VECTOR * position
);
```

Parameters

`position`

Address of a variable to receive the 3d position of the object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Geometry::setPosition`
- `FMOD_VECTOR`
Firelight Technologies FMOD Ex
Geometry::getRotation

Retrieves the orientation of the geometry object.

C++ Syntax

```cpp
FMOD_RESULT Geometry::getRotation(
    FMOD_VECTOR * forward,
    FMOD_VECTOR * up
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetRotation(
    FMOD_GEOMETRY * geometry,
    FMOD_VECTOR * forward,
    FMOD_VECTOR * up
);
```

Parameters

*forward*

Address of a variable that receives the forwards orientation of the geometry object. Specify 0 or NULL to ignore.

*up*

Address of a variable that receives the upwards orientation of the geometry object. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

See remarks in System::set3DListenerAttributes for more description on forward and up vectors.

See Also

- Geometry::setRotation
- System::set3DLlistenerAttributes
- FMOD_VECTOR
Firelight Technologies FMOD Ex
Geometry::getScale

Retrieves the relative scale vector of the geometry object. An object can be scaled/warped in all 3 dimensions separately using the vector.

C++ Syntax

```c++
FMOD_RESULT Geometry::getScale(
    FMOD_VECTOR * scale
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetScale(
    FMOD_GEOMETRY * geometry,
    FMOD_VECTOR * scale
);
```

Parameters

`scale`

Address of a variable to receive the scale vector of the object. Default = 1.0, 1.0, 1.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Geometry::setScale`
- `FMOD_VECTOR`
Firelight Technologies FMOD Ex
Geometry::getUserData

Retrieves the user value that was set by calling the Geometry::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT Geometry::getUserData(
    void ** userdata
);```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_GetUserData(
    FMOD_GEOMETRY * geometry,
    void ** userdata
);```

Parameters

`userdata`

Address of a pointer that receives the data specified with the Geometry::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- Geometry::setUserData
Firelight Technologies FMOD Ex
Geometry::release

Frees a geometry object and releases its memory.

C++ Syntax

```cpp
FMOD_RESULT Geometry::release();
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_Release(FMOD_GEOMETRY * geometry);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Firelight Technologies FMOD Ex
Geometry::save

Saves the geometry object as a serialized binary block, to a user memory buffer. This can then be saved to a file if required and loaded later with System::loadGeometry.

C++ Syntax

```cpp
FMOD_RESULT Geometry::save(
    void * data,
    int * datasize
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_Save(
    FMOD_GEOMETRY * geometry,
    void * data,
    int * datasize
);
```

Parameters

*data*

Address of a variable to receive the serialized geometry object. Specify 0 or NULL to have the datasize parameter return the size of the memory required for this saved object.

*datasize*

Address of a variable to receive the size in bytes required to save this object when 'data' parameter is 0 or NULL.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the
**Remarks**

To use this function you will normally need to call it twice. Once to get the size of the data, then again to write the data to your pointer.

**See Also**

- [System::loadGeometry](#)
- [System::createGeometry](#)
Firelight Technologies FMOD Ex
Geometry::setActive

Enables or disables an object from being processed in the geometry engine.

C++ Syntax

```cpp
FMOD_RESULT Geometry::setActive(
    bool active
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_SetActive(
    FMOD_GEOMETRY * geometry,
    FMOD_BOOL active
);
```

Parameters

active

true = active, false = not active. Default = true.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Geometry::getActive
Firelight Technologies FMOD Ex
**Geometry::setPolygonAttributes**

Sets individual attributes for each polygon inside a geometry object.

### C++ Syntax

```cpp
FMOD_RESULT Geometry::setPolygonAttributes(
    int index,
    float directocclusion,
    float reverbocclusion,
    bool doublesided
);
```

### C Syntax

```c
FMOD_RESULT FMOD_Geometry_SetPolygonAttributes(
    FMOD_GEOMETRY * geometry,
    int index,
    float directocclusion,
    float reverbocclusion,
    FMOD_BOOL doublesided
);
```

### Parameters

- **index**
  
  Polygon index inside the object.

- **directocclusion**
  
  Occlusion value from 0.0 to 1.0 which affects volume or audible frequencies. 0.0 = The polygon does not occlude volume or audible frequencies (sound will be fully audible), 1.0 = The polygon fully occludes (sound will be silent).

- **reverbocclusion**
  
  Occlusion value from 0.0 to 1.0 which affects the reverb mix. 0.0 = The polygon does not occlude reverb (reverb reflections still travel through this polygon), 1.0 = The polyfully fully occludes reverb (reverb reflections will be silent through
this polygon).

doublesided

Description of polygon if it is double sided or single sided. true = polygon is double sided, false = polygon is single sided, and the winding of the polygon (which determines the polygon's normal) determines which side of the polygon will cause occlusion.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- Geometry::getPolygonAttributes
- Geometry::getNumPolygons
Firelight Technologies FMOD Ex
**Geometry::setPolygonVertex**

Alters the position of a polygon's vertex inside a geometry object.

**C++ Syntax**

```cpp
FMOD_RESULT Geometry::setPolygonVertex(
    int index,
    int vertexindex,
    const FMOD_VECTOR * vertex
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Geometry_SetPolygonVertex(
    FMOD_GEOMETRY * geometry,
    int index,
    int vertexindex,
    const FMOD_VECTOR * vertex
);
```

**Parameters**

*index*

Polygon index. This must be in the range of 0 to `Geometry::getNumPolygons` minus 1.

*vertexindex*

Vertex index inside the polygon. This must be in the range of 0 to `Geometry::getPolygonNumVertices` minus 1.

*vertex*

Address of an `FMOD_VECTOR` which holds the new vertex location.

**Return Values**
If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

**Note!** There may be some significant overhead with this function as it may cause some reconfiguration of internal data structures used to speed up sound-ray testing.
You may get better results if you want to modify your object by using
**Geometry::setPosition**, **Geometry::setScale** and **Geometry::setRotation**.

**See Also**

- **Geometry::getPolygonNumVertices**
- **Geometry::getPolygonNumVertices**
- **Geometry::setPosition**
- **Geometry::setScale**
- **Geometry::setRotation**
- **Geometry::getNumPolygons**
- **FMOD_VECTOR**
Firelight Technologies FMOD Ex
Geometry::setPosition

Sets the position of the object in world space, which is the same space FMOD sounds and listeners reside in.

C++ Syntax

```cpp
FMOD_RESULT Geometry::setPosition(
    const FMOD_VECTOR * position
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_SetPosition(
    FMOD_GEOMETRY * geometry,
    const FMOD_VECTOR * position
);
```

Parameters

*position*

Pointer to a vector containing the 3d position of the object.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- Geometry::getPosition
- Geometry::setRotation
- Geometry::setScale
• FMOD_VECTOR
Firelight Technologies FMOD Ex
Geometry::setRotation

Sets the orientation of the geometry object.

C++ Syntax

```cpp
FMOD_RESULT Geometry::setRotation(
    const FMOD_VECTOR * forward,
    const FMOD_VECTOR * up
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_SetRotation(
    FMOD_GEOMETRY * geometry,
    const FMOD_VECTOR * forward,
    const FMOD_VECTOR * up
);
```

Parameters

*forward*

The forwards orientation of the geometry object. This vector must be of unit length and perpendicular to the up vector. You can specify 0 or NULL to not update the forwards orientation of the geometry object.

*up*

The upwards orientation of the geometry object. This vector must be of unit length and perpendicular to the forwards vector. You can specify 0 or NULL to not update the upwards orientation of the geometry object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

See remarks in System::set3DListenerAttributes for more description on forward and up vectors.

See Also

- Geometry::getRotation
- System::set3DListenerAttributes
- FMOD_VECTOR
Firelight Technologies FMOD Ex
Geometry::setScale

Sets the relative scale vector of the geometry object. An object can be scaled/warped in all 3 dimensions separately using the vector without having to modify polygon data.

C++ Syntax

```cpp
FMOD_RESULT Geometry::setScale(
    const FMOD_VECTOR * scale
);
```

C Syntax

```c
FMOD_RESULT FMOD_Geometry_SetScale(
    FMOD_GEOMETRY * geometry,
    const FMOD_VECTOR * scale
);
```

Parameters

scale

The scale vector of the object. Default = 1.0, 1.0, 1.0.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- Geometry::getScale
- Geometry::setRotation
• `Geometry::setPosition`
• `FMOD_VECTOR`
Firelight Technologies FMOD Ex
**Geometry::setUserData**

Sets a user value that the Geometry object will store internally. Can be retrieved with `Geometry::getUserData`.

**C++ Syntax**

```cpp
FMOD_RESULT Geometry::setUserData(
    void * userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Geometry_SetUserData(
    FMOD_GEOMETRY * geometry,
    void * userdata
);
```

**Parameters**

*userdata*

Address of user data that the user wishes stored within the Geometry object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.

If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.

It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using Geometry::getUserData would help in the identification of the object.

**See Also**

- Geometry::getUserData
Firelight Technologies FMOD Ex
Reverb Interface

Reverb::get3DAttributes  Reverb::getActive
Reverb::getMemoryInfo
Reverb::getProperties
Reverb::getUserData
Reverb::release
Reverb::set3DAttributes
Reverb::setActive
Reverb::setProperties
Reverb::setUserData
Firelight Technologies FMOD Ex
Reverb::get3DAttributes

Retrieves the 3d attributes of a Reverb object.

C++ Syntax

```cpp
FMOD_RESULT Reverb::get3DAttributes(
    FMOD_VECTOR * position,
    float * mindistance,
    float * maxdistance
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb_Get3DAttributes(
    FMOD_REVERB * reverb,
    FMOD_VECTOR * position,
    float * mindistance,
    float * maxdistance
);
```

Parameters

*position*

Address of a variable that will receive the 3d position of the center of the reverb in 3d space. Default = \{ 0,0,0 \}.

*mindistance*

Address of a variable that will receive the distance from the centerpoint that the reverb will have full effect at. Default = 0.0.

*maxdistance*

Address of a variable that will receive the distance from the centerpoint that the reverb will not have any effect. Default = 0.0.
Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

The 3D reverb object is a sphere having 3D attributes (position, minimum distance, maximum distance) and reverb properties.
The properties and 3D attributes of all reverb objects collectively determine, along with the listener's position, the settings of and input gains into a single 3D reverb DSP.
Please note that this only applies to software channels. When the listener is within the sphere of effect of one or more 3d reverbs, the listener's 3D reverb properties are a weighted combination of such 3d reverbs. When the listener is outside all of the reverbs, the 3D reverb setting is set to the default ambient reverb setting.

See Also

- Reverb::set3DAttributes
- System::createReverb
Firelight Technologies FMOD Ex
**Reverb::getActive**

Retrieves the active state of the reverb object.

**C++ Syntax**

```cpp
FMOD_RESULT Reverb::getActive(
    bool * active
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Reverb_GetActive(
    FMOD_REVERB * reverb,
    FMOD_BOOL * active
);
```

**Parameters**

*active*

Address of a variable to receive the current active state of the reverb object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- [Reverb::setActive](#)
- [System::createReverb](#)
Firelight Technologies FMOD Ex
Reverb::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT Reverb::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb_GetMemoryInfo(
    FMOD_REVERB * reverb,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

**memorybits**

Memory usage bits for FMOD Ex. See [FMOD_MEMBITS](https://www.openal-rev.org/docs/fmod.html#membits).

**event_memorybits**

Memory usage bits for FMOD Event System. See [FMOD_EVENT_MEMBITS](https://www.openal-rev.org/docs/fmod.html#event_membits).

**memoryused**

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

**memoryused_details**
Optional. Specify 0 to ignore. Address of a user-allocated `FMOD_MEMORY_USAGE_DETAILS` structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See `System::getMemoryInfo` for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGE_DETAILS`
- `System::getMemoryInfo`
Firelight Technologies FMOD Ex
Reverb::getProperties

Retrieves the current reverb environment.

C++ Syntax

```cpp
FMOD_RESULT Reverb::getProperties(
    FMOD_REVERB_PROPERTIES * properties
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb_GetProperties(
    FMOD_REVERB * reverb,
    FMOD_REVERB_PROPERTIES * properties
);
```

Parameters

properties

Address of a variable that receives the current reverb environment description.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- `Reverb::setProperties`
- `System::createReverb`
Firelight Technologies FMOD Ex
Reverb::getUserData

Retrieves the user value that was set by calling the Reverb::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT Reverb::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb_GetUserData(
    FMOD_REVERB * reverb,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the Reverb::setUserData function.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- Reverb::setUserData
Firelight Technologies FMOD Ex
Reverb::release

Releases the memory for a reverb object and makes it inactive.

C++ Syntax

```cpp
FMOD_RESULT Reverb::release();
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb_Release(FMOD_REVERB * reverb);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

If no reverb objects are created, the ambient reverb will be the only audible reverb. By default this ambient reverb setting is set to OFF.

See Also

- [System::createReverb](#)
- [System::setReverbAmbientProperties](#)
Firelight Technologies FMOD Ex
Reverb::set3DAttributes

Sets the 3d properties of a 'virtual' reverb object.

C++ Syntax

```cpp
FMOD_RESULT Reverb::set3DAttributes(
    const FMOD_VECTOR * position,
    float mindistance,
    float maxdistance
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb_Set3DAttributes(
    FMOD_REVERB * reverb,
    const FMOD_VECTOR * position,
    float mindistance,
    float maxdistance
);
```

Parameters

position

Pointer to a vector containing the 3d position of the center of the reverb in 3d space. Default = { 0,0,0 }.

mindistance

The distance from the centerpoint that the reverb will have full effect at. Default = 0.0.

maxdistance

The distance from the centerpoint that the reverb will not have any effect. Default = 0.0.
Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

The 3D reverb object is a sphere having 3D attributes (position, minimum distance, maximum distance) and reverb properties. The properties and 3D attributes of all reverb objects collectively determine, along with the listener's position, the settings of and input gains into a single 3D reverb DSP.
Please note that this only applies to software channels. When the listener is within the sphere of effect of one or more 3d reverbs, the listener's 3D reverb properties are a weighted combination of such 3d reverbs. When the listener is outside all of the reverbs, the 3D reverb setting is set to the default ambient reverb setting.

See Also

- Reverb::get3DAttributes
- System::createReverb
Firelight Technologies FMOD Ex
**Reverb::setActive**

Disables or enables a reverb object so that it does or does not contribute to the 3d scene.

**C++ Syntax**

```cpp
FMOD_RESULT Reverb::setActive(
    bool active
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Reverb_SetActive(
    FMOD_REVERB * reverb,
    FMOD_BOOL active
);
```

**Parameters**

*active*

true = active, false = not active. Default = true.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- [Reverb::setActive](#)
- [System::createReverb](#)
Reverb::setProperties

Sets reverb parameters for the current reverb object. Reverb parameters can be set manually, or automatically using the pre-defined presets given in the fmod.h header.

C++ Syntax

```cpp
FMOD_RESULT Reverb::setProperties(
    const FMOD_REVERB_PROPERTIES * properties
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb_SetProperties(
    FMOD_REVERB * reverb,
    const FMOD_REVERB_PROPERTIES * properties
);
```

Parameters

`properties`

Address of an FMOD_REVERB_PROPERTIES structure which defines the attributes for the reverb.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- FMOD_REVERB_PROPERTIES
• Reverb::getProperties
• System::createReverb
Firelight Technologies FMOD Ex
Reverb::setUserData

Sets a user value that the Reverb object will store internally. Can be retrieved with Reverb::getUserData.

C++ Syntax

```cpp
FMOD_RESULT Reverb::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb_SetUserData(
    FMOD_REVERB * reverb,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the Reverb object.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using Reverb::getUserData would help in the identification of the object.

See Also

- Reverb::getUserData
Firelight Technologies FMOD Ex
Functions

Debug_GetLevel Debug_SetLevel
File_GetDiskBusy
File_SetDiskBusy
Memory_GetStats
Memory_Initialize
System_Create
Firelight Technologies FMOD Ex
Debug_GetLevel

Retrieves the current debug logging level.

C++ Syntax

```cpp
FMOD_RESULT Debug_GetLevel( 
    FMOD_DEBUGLEVEL * level
);
```

C Syntax

```c
FMOD_RESULT FMOD_Debug_GetLevel( 
    FMOD_DEBUGLEVEL * level
);
```

Parameters

`level`

Address of a variable to receive current debug level.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This only has an effect with 'logging' versions of FMOD Ex. For example on windows it must be via fmodexL.dll, not fmodex.dll. On Xbox it would be fmodxboxL.lib not fmodxbox.lib. `FMOD_ERR_UNSUPPORTED` will be returned on non logging versions of FMOD Ex (ie full release).
See Also

- Debug_SetLevel
- FMOD_DEBUGLEVEL
Firelight Technologies FMOD Ex
**Debug_SetLevel**

Sets the level of debug logging to the tty / output for logging versions of FMOD Ex.

**C++ Syntax**

```cpp
FMOD_RESULT Debug_SetLevel(  
    FMOD_DEBUGLEVEL   level
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Debug_SetLevel(  
    FMOD_DEBUGLEVEL   level
);
```

**Parameters**

*level*

Logging level to set.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

This only has an effect with 'logging' versions of FMOD Ex. For example on windows it must be via fmodexL.dll, not fmodex.dll. On Xbox it would be fmodxboxL.lib not fmodxbox.lib. **FMOD_ERR_UNSUPPORTED** will be returned on non logging versions of
FMOD Ex (ie full release).

See Also

- [Debug_GetLevel](#)
- [FMOD_DEBUGLEVEL](#)
Firelight Technologies FMOD Ex
File_GetDiskBusy

Information function to retrieve the state of fmod's disk access.

C++ Syntax

```cpp
FMOD_RESULT File_GetDiskBusy(
    int * busy
);
```

C Syntax

```c
FMOD_RESULT FMOD_File_GetDiskBusy(
    int * busy
);
```

**Parameters**

`busy`

Address of an integer to receive the busy state of the disk at the current time.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Do not use this function to synchronize your own reads with, as due to timing, you might call this function and it says false = it is not busy, but the split second after call this function, internally FMOD might set it to busy. Use `File_SetDiskBusy` for proper mutual exclusion as it uses semaphores.
See Also

- File_SetDiskBusy
Firelight Technologies FMOD Ex
**File_SetDiskBusy**

Mutex function to synchronize user file reads with FMOD's file reads. This function tells fmod that you are using the disk so that it will block until you are finished with it.

This function also blocks if FMOD is already using the disk, so that you cannot do a read at the same time FMOD is reading.

**C++ Syntax**

```cpp
FMOD_RESULT File_SetDiskBusy(
    int busy
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_File_SetDiskBusy(
    int busy
);
```

**Parameters**

*busy*

1 = you are about to perform a disk access. 0 = you are finished with the disk.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.

If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

Use this function as a wrapper around your own file reading functions if you...
want to do simultaneous file reading while FMOD is also reading, ie

```c
FMOD_File_SetDiskBusy(1);
myfread(...);
FMOD_File_SetDiskBusy(0);
```

Warning! This is a critical section internally. If you do not match your busy =
true with a busy = false your program may hang!
If you forget to set diskbusy to false it will stop FMOD from reading from the
disk.

**See Also**

- [File_GetDiskBusy](#)
Firelight Technologies FMOD Ex
Memory_GetStats

Returns information on the memory usage of FMOD. This is useful for determining a fixed memory size to make FMOD work within for fixed memory machines such as consoles.

C++ Syntax

```cpp
FMOD_RESULT Memory_GetStats(
    int * currentallocated,
    int * maxallocated,
    bool blocking
);
```

C Syntax

```c
FMOD_RESULT FMOD_Memory_GetStats(
    int * currentallocated,
    int * maxallocated,
    FMOD_BOOL blocking
);
```

Parameters

`currentallocated`

Address of a variable that receives the currently allocated memory at time of call. Optional. Specify 0 or NULL to ignore.

`maxallocated`

Address of a variable that receives the maximum allocated memory since `System::init` or `Memory.Initialize`. Optional. Specify 0 or NULL to ignore.

`blocking`

Boolean indicating whether to favour speed or accuracy. Specifying true for this parameter will flush the DSP network to make sure all queued allocations happen immediately, which can be costly.
Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Note that if using FMOD::Memory_Initialize, the memory usage will be slightly higher than without it, as FMOD has to have a small amount of memory overhead to manage the available memory.

See Also

- System::init
- Memory_Initialize
Firelight Technologies FMOD Ex
Memory_Initialize

Specifies a method for FMOD to allocate memory, either through callbacks or its own internal memory management. You can also supply a pool of memory for FMOD to work with and it will do so with no extra calls to malloc or free.

C++ Syntax

```cpp
FMOD_RESULT Memory_Initialize(
    void * poolmem,
    int poollen,
    FMOD_MEMORY_ALLOCCALLBACK useralloc,
    FMOD_MEMORY_REALLOCCALLBACK userrealloc,
    FMOD_MEMORY_FREECALLBACK userfree,
    FMOD_MEMORY_TYPE memtypeflags
);
```

C Syntax

```c
FMOD_RESULT FMOD_Memory_Initialize(
    void * poolmem,
    int poollen,
    FMOD_MEMORY_ALLOCCALLBACK useralloc,
    FMOD_MEMORY_REALLOCCALLBACK userrealloc,
    FMOD_MEMORY_FREECALLBACK userfree,
    FMOD_MEMORY_TYPE memtypeflags
);
```

Parameters

`poolmem`

If you want a fixed block of memory for FMOD to use, pass it in here. Specify the length in `poollen`. Specifying NULL doesn't use internal management and it relies on callbacks.

`poollen`

Length in bytes of the pool of memory for FMOD to use specified in. Specifying 0 turns off internal memory management and relies purely on callbacks. Length must be a multiple of 512.
useralloc

Only supported if pool is NULL. Otherwise it overrides the FMOD internal calls to alloc. Compatible with ansi malloc().

userrealloc

Only supported if pool is NULL. Otherwise it overrides the FMOD internal calls to realloc. Compatible with ansi realloc().

userfree

Only supported if pool is NULL. Otherwise it overrides the FMOD internal calls to free. Compatible with ansi free().

memtypeflags

FMOD_MEMORY_TYPE flags you wish to receive through your memory callbacks. See FMOD_MEMORY_TYPE.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function is useful for systems that want FMOD to use their own memory management or for fixed memory devices such as Xbox360 that don't want any allocations occurring out of their control causing fragmentation or unpredictable overflows in a tight memory space.

FMOD only does allocation when creating streams, music or samples and the System:::init stage. It never allocates or deallocates memory during the course of runtime processing. To find out the required fixed size the user can call
Memory_Initialize with an overly large pool size (or no pool) and find out the maximum RAM usage at any one time with Memory_GetStats.

FMOD behaves differently based on what you pass into this function in 3 different combinations. For example:

FMOD::Memory_Initialize(NULL, 0, NULL, NULL, NULL);  // F
FMOD::Memory_Initialize(NULL, 0, myalloc, myrealloc, myfree); // C
FMOD::Memory_Initialize(ptr, len, NULL, NULL, NULL);  // U

Callbacks and memory pools cannot be combined. If a memory pool is provided by the user, FMOD accesses that pool using its own memory management scheme. FMOD's internal memory management scheme is extremely efficient and also faster than the standard C malloc and free.

**NOTE!** Your memory callbacks must be threadsafe otherwise unexpected behaviour may occur. FMOD calls memory allocation functions from other threads (such as the asynchronous loading thread used when you specify FMOD_NONBLOCKING) and sometimes from the mixer thread.

**NOTE!** If you specify a fixed size pool that is too small, FMOD will return FMOD_ERR_MEMORY when the limit of the fixed size pool is exceeded. At this point, it's possible that FMOD may become unstable. To maintain stability, do not allow FMOD to run out of memory.

See Also

- FMOD_MEMORY_ALLOCCALLBACK
- FMOD_MEMORY_REALLOCCALLBACK
- FMOD_MEMORY_FREECALLBACK
- Memory_GetStats
- System::close
Firelight Technologies FMOD Ex
System_Create

FMOD System creation function. This must be called to create an FMOD System object before you can do anything else. Use this function to create 1, or multiple instances of FMOD System objects.

C++ Syntax

```cpp
FMOD_RESULT System_Create(
    FMOD::System ** system
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_Create(
    FMOD_SYSTEM ** system
);
```

Parameters

`system`

Address of a pointer that receives the new FMOD System object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Use `System::release` to free a system object.
See Also

- System::init
- System::release
Firelight Technologies FMOD Ex
Callbacks

FMOD_3D_ROLLOFFCALLBACK
FMOD_CODEC_CLOSECALLBACK
FMOD_CODEC_GETLENGTHCALLBACK
FMOD_CODEC_GETPOSITIONCALLBACK
FMOD_CODEC_METADATACALLBACK
FMOD_CODEC_OPENCALLBACK
FMOD_CODEC_READCALLBACK
FMOD_CODEC_SETPOSITIONCALLBACK
FMOD_CODEC_SOUNDCREATECALLBACK
FMOD_DSP_CREATECALLBACK
FMOD_DSP_DIALOGCALLBACK
FMOD_DSP_GETPARAMCALLBACK
FMOD_DSP_READCALLBACK
FMOD_DSP_RELEASECALLBACK
FMOD_DSP_RESETCALLBACK
FMOD_DSP_SETPARAMCALLBACK
FMOD_DSP_SETPOSITIONCALLBACK
FMOD_FILEASYNCCANCELCALLBACK
FMOD_FILEASYNCREADCALLBACK
FMOD_FILE_CLOSECALLBACK
FMOD_FILE_OPENCALLBACK
FMOD_FILE_READCALLBACK
FMOD_FILE_SEEKCALLBACK
FMOD_MEMORY_ALLOCATECALLBACK
FMOD_MEMORY_FREECALLBACK
FMOD_MEMORY_REALLOCATECALLBACK
FMOD_OUTPUT_CLOSECALLBACK
FMOD_OUTPUT_GETDRIVERCAPSCALLBACK
FMOD_OUTPUT_GETDRIVERNAMECALLBACK
FMOD_OUTPUT_GETHANDELDCALLBACK
FMOD_OUTPUT_GETNUMDRIVERSCALLBACK
FMOD_OUTPUT_GETPOSITIONCALLBACK
FMOD_OUTPUT_INITCALLBACK
FMOD_OUTPUT_LOCKCALLBACK
FMOD_OUTPUT_READFROMMIXER
FMOD_OUTPUT_UNLOCKCALLBACK
FMOD_OUTPUT_UPDATECALLBACK
FMOD_SOUND_NONBLOCKCALLBACK
FMOD_SOUND_PCMREADCALLBACK
FMOD_SOUND_PCMSETPOSCALLBACK
FMOD_SYSTEM_CALLBACK
Firelight Technologies FMOD Ex
**FMOD_3D_ROLLOFFCALLBACK**

Callback for system wide 3d channel volume calculation which overrides fmod's internal calculation code.

**C/C++ Syntax**

```c
float F_CALLBACK FMOD_3D_ROLLOFFCALLBACK(
    FMOD_CHANNEL * channel,
    float distance
);
```

**Parameters**

*channel*

Pointer to a channel handle.

*distance*

Distance in units (meters by default).

**Return Values**

**Remarks**

**C++ Users.** Cast `FMOD_CHANNEL *` to `FMOD::Channel *` inside the callback and use as normal.

NOTE: When using the event system, call `Channel::getUserData` to get the event instance handle of the event that spawned the channel in question.

**See Also**
- `System::set3DRolloffCallback`
- `System::set3DListenerAttributes`
- `System::get3DListenerAttributes`
- `Channel::getUserData`
Firelight Technologies FMOD Ex
**FMOD_CHANNEL_CALLBACK**

Callback for channel events.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_CHANNEL_CALLBACK(
    FMOD_CHANNEL * channel,
    FMOD_CHANNEL_CALLBACKTYPE type,
    void * commanddata1,
    void * commanddata2
);
```

**Parameters**

*channel*

Pointer to a channel handle.

*type*

The type of callback. Refer to [FMOD_CHANNEL_CALLBACKTYPE](#).

*commanddata1*

The first callback type specific data generated by the callback. See remarks for meaning.

*commanddata2*

The second callback type specific data generated by the callback. See remarks for meaning.

**Return Values**

If the function succeeds then the return value is [FMOD_OK](#).
If the function fails then the return value will be one of the values defined in the
FMOD_RESULT enumeration.

Remarks

C++ Users. Cast FMOD_CHANNEL * to FMOD::Channel * inside the callback and use as normal.

'commanddata1' and 'commanddata2' meanings. These 2 values are set by the callback depending on what is happening in the callback and the type of callback.

- **FMOD_CHANNEL_CALLBACKTYPE_END**
  
  commanddata1: Always 0.
  
  commanddata2: Always 0.

- **FMOD_CHANNEL_CALLBACKTYPE_VIRTUALVOICE**
  
  commanddata1: (cast to int) 0 when voice is swapped from emulated to real. 1 when voice is swapped from real to emulated.
  
  commanddata2: Always 0.

- **FMOD_CHANNEL_CALLBACKTYPE_SYNCPOINT**
  
  commanddata1: (cast to int) The index of the sync point. Use Sound::getSyncPointInfo to retrieve the sync point's attributes.
  
  commanddata2: Always 0.

- **FMOD_CHANNEL_CALLBACKTYPE_OCCLUSION**
  
  commanddata1: (cast to float *) pointer to a floating point direct value that can be read (dereferenced) and modified after the geometry engine has calculated it for this channel.
  
  commanddata2: (cast to float *) pointer to a floating point reverb value that can be read (dereferenced) and modified after the geometry engine has calculated it for this channel.

Note! Currently the user must call System::update for these callbacks to trigger!

See Also

- Channel::setCallback
- FMOD_CHANNEL_CALLBACKTYPE
• System::update
Firelight Technologies FMOD Ex
**FMOD_CODEC_CLOSECALLBACK**

Close callback for the codec for when FMOD tries to close a sound using this codec.

This is the callback any codec related memory is freed, and things are generally de-initialized / shut down for the codec.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_CODEC_CLOSECALLBACK(
    FMOD_CODEC_STATE * codec_state
);
```

**Parameters**

*codec_state*

Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.

**See Also**
- FMOD_CODEC_STATE
- FMOD_CODEC_DESCRIPTION
- FMOD_CODEC_OPENCALLBACK
- FMOD_CODEC_READCALLBACK
- FMOD_CODEC_GETLENGTHCALLBACK
- FMOD_CODEC_SETPOSITIONCALLBACK
- FMOD_CODEC_GETPOSITIONCALLBACK
- FMOD_CODEC_SOUNDCREATECALLBACK

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
Callback to return the length of the song in whatever format required when Sound::getLength is called.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_CODEC_GETLENGTHCALLBACK(
    FMOD_CODEC_STATE * codec_state,
    unsigned int * length,
    FMOD_TIMEUNIT lengthtype
);
```

**Parameters**

*codec_state*

Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

*length*

Address of a variable that is to receive the length of the sound determined by the format specified in the lengthtype parameter.

*lengthtype*

Timeunit type of length to return. This will be one of the timeunits supplied by the codec author in the `FMOD_CODEC_DESCRIPTION` structure.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.

See Also

- **FMOD_TIMEUNIT**
- **FMOD_CODEC_STATE**
- **FMOD_CODEC_DESCRIPTION**
- **FMOD_CODEC_OPENCALLBACK**
- **FMOD_CODEC_CLOSECALLBACK**
- **FMOD_CODEC_READCALLBACK**
- **FMOD_CODEC_SETPOSITIONCALLBACK**
- **FMOD_CODEC_GETPOSITIONCALLBACK**
- **FMOD_CODEC_SOUNDCREATECALLBACK**
Firelight Technologies FMOD Ex
Tell callback for the codec for when FMOD tries to get the current position within the with `Channel::getPosition`.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_CODEC_GETPOSITIONCALLBACK(
    FMOD_CODEC_STATE * codec_state,
    unsigned int * position,
    FMOD_TIMEUNIT postype
);
```

**Parameters**

- **codec_state**
  
  Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

- **position**
  
  Address of a variable to receive the current position in the codec based on the timeunit specified in the postype parameter.

- **postype**
  
  Timeunit type of the position parameter that is requested. This will be one of the timeunits supplied by the codec author in the `FMOD_CODEC_DESCRIPTION` structure.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.

See Also

- [Channel::getPosition](#)
- [FMOD_CODEC_STATE](#)
- [FMOD_CODEC_DESCRIPTION](#)
- [FMOD_CODEC_OPENCALLBACK](#)
- [FMOD_CODEC_CLOSECALLBACK](#)
- [FMOD_CODEC_READCALLBACK](#)
- [FMOD_CODEC_GETLENGTHCALLBACK](#)
- [FMOD_CODEC_SETPOSITIONCALLBACK](#)
- [FMOD_CODEC_SOUNDCALLBACK](#)
Firelight Technologies FMOD Ex
FMOD_CODEC_METADATACALLBACK

Callback for sounds that have their

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_CODEC_METADATACALLBACK(
    FMOD_CODEC_STATE * codec_state,
    FMOD_TAGTYPE type,
    char * name,
    void * data,
    unsigned int datalen,
    FMOD_TAGDATATYPE datatype,
    int unique
);
```

Parameters

`codec_state`

Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

`type`

Source of tag being updated, ie id3v2 or ogg vorbis tag for example. See `FMOD_TAGDATATYPE`.

`name`

Name of the tag being updated.

`data`

Contents of tag.

`datalen`

Length of the tag data in bytes.
The `datatype` parameter specifies the data type of the tag. It can be Binary, string, or Unicode, among others. For more information, see the `FMOD_TAGDATATYPE` enumeration.

The `unique` parameter indicates whether the tag (determined by the name) being updated is the only one of its type. If it is false, there are multiple versions of this tag with the same name.

### Return Values

If the function succeeds, the return value is `FMOD_OK`. If the function fails, the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

### Remarks

This callback is usually called from sounds that can update their metadata / tag info at runtime. Such a sound could be an internet SHOUTcast / Icecast stream, for example.

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`.

### See Also

- `FMOD_CODEC_STATE`
- `FMOD_CODEC_DESCRIPTION`
- `FMOD_CODEC_OPENCALLBACK`
- `FMOD_CODEC_CLOSECALLBACK`
- `FMOD_CODEC_READCALLBACK`
- `FMOD_CODEC_GETLENGTHCALLBACK`
- `FMOD_CODEC_SETPOSITIONCALLBACK`
- **FMOD_CODEC_GETPOSITIONCALLBACK**
- **FMOD_CODEC_SOUNDCREATECALLBACK**
- **FMOD_TAGDATATYPE**
Firelight Technologies FMOD Ex
FMOD_CODEC_OPENCALLBACK

Open callback for the codec for when FMOD tries to open a sound using this codec. This is the callback the file format check is done in, codec related memory is allocated, and things are generally initialized / set up for the codec.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_CODEC_OPENCALLBACK(
    FMOD_CODEC_STATE * codec_state,
    FMOD_MODE usermode,
    FMOD_CREATESOUNDEXINFO * userexinfo
);
```

Parameters

codec_state

Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

usermode

Mode that the user supplied via `System::createSound`. This is informational and can be ignored, or used if it has relevance to your codec.

userexinfo

Extra info structure that the user supplied via `System::createSound`. This is informational and can be ignored, or used if it has relevance to your codec.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

The usermode and userexinfo parameters tell the codec what was passed in by the user. Generally these can be ignored, as the file format usually determines the format and frequency of the sound.

If you have a flexible format codec (ie you don't mind what output format your codec writes to), you might want to use the parameter that was passed in by the user to specify the output sound format / frequency. For example if you normally create a codec that is always 32bit floating point, the user might supply 16bit integer to save memory, so you could use this information to decode your data to this format instead of the original default format.

Read and seek within the file using the 'fileread' and 'fileseek' members of the FMOD_CODEC codec that is passed in.

Note: **DO NOT USE YOUR OWN FILESYSTEM.**
The reasons for this are:

- The user may have set their own file system via user filesystem callbacks.
- FMOD allows file reading via disk, memory and TCP/IP. If you use your own file routines you will lose this ability.

**Important!** FMOD will ping all codecs trying to find the right one for the file the user has passed in. Make sure the first line of your codec open is a FAST format check. Ie it reads an identifying string, checks it and returns an error **FMOD_ERR_FORMAT** if it is not found.
There may be a lot of codecs loaded into FMOD, so you don't want yours slowing down the **System::createSound** call because it is inneficient in determining if it is the right format or not.

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.

**See Also**
- **System::createSound**
- **FMOD_CREATESOUNDEXINFO**
- **FMOD_CODEC_STATE**
- **FMOD_CODEC_DESCRIPTION**
- **FMOD_CODEC_CLOSECALLBACK**
- **FMOD_CODEC_READCALLBACK**
- **FMOD_CODEC_GETLENGTHCALLBACK**
- **FMOD_CODEC_SETPOSITIONCALLBACK**
- **FMOD_CODEC_GETPOSITIONCALLBACK**
- **FMOD_CODEC_SOUNDCREATECALLBACK**

*Version 4.44.07 Built on Feb 11, 2013*
Firelight Technologies FMOD Ex
FMOD_CODEC_READCALLBACK

Read callback for the codec for when FMOD tries to read some data from the file to the destination format (format specified in the open callback).

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_CODEC_READCALLBACK(
    FMOD_CODEC_STATE * codec_state,
    void * buffer,
    unsigned int sizebytes,
    unsigned int * bytesread
);
```

Parameters

codec_state

Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

buffer

Buffer to read PCM data to. Note that the format of this data is the format described in FMOD_CODEC_WAVEFORMAT.

sizebytes

Number of bytes to read

bytesread

Number of bytes actually read

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

\textbf{Remarks}

If you cannot read number of bytes requested, simply return \texttt{FMOD\_OK} and give \texttt{bytesread} the number of bytes you read.
Read and seek within the file using the 'fileread' and 'fileseek' members of the \texttt{FMOD\_CODEC} codec that is passed in.
Note: \textbf{DO NOT USE YOUR OWN FILESYSTEM}.
The reasons for this are:

- The user may have set their own file system via user filesystem callbacks.
- FMOD allows file reading via disk, memory and TCP/IP. If you use your own file routines you will lose this ability.

Remember to return \texttt{FMOD\_OK} at the bottom of the function, or an appropriate error code from \texttt{FMOD\_RESULT}.

\textbf{See Also}

- \texttt{FMOD\_CODEC\_STATE}
- \texttt{FMOD\_CODEC\_DESCRIPTION}
- \texttt{FMOD\_CODEC\_OPENCALLBACK}
- \texttt{FMOD\_CODEC\_CLOSECALLBACK}
- \texttt{FMOD\_CODEC\_GETLENGTHCALLBACK}
- \texttt{FMOD\_CODEC\_SETPOSITIONCALLBACK}
- \texttt{FMOD\_CODEC\_GETPOSITIONCALLBACK}
- \texttt{FMOD\_CODEC\_SOUNDCREATECALLBACK}
Firelight Technologies FMOD Ex
FMOD_CODEC_SETPOSITIONCALLBACK

Seek callback for the codec for when FMOD tries to seek within the file with Channel::setPosition.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_CODEC_SETPOSITIONCALLBACK(
    FMOD_CODEC_STATE * codec_state,
    int subsound,
    unsigned int position,
    FMOD_TIMEUNIT postype
);
```

Parameters

codec_state

Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

subsound

Subsound within which to seek.

position

Position to seek to in the sound based on the timeunit specified in the postype parameter.

postype

Timeunit type of the position parameter. This will be one of the timeunits supplied by the codec author in the FMOD_CODEC_DESCRIPTION structure.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

Read and seek within the file using the 'fileread' and 'fileseek' members of the **FMOD_CODEC** codec that is passed in.

*Note: DO NOT USE YOUR OWN FILESYSTEM.*
The reasons for this are:

- The user may have set their own file system via user filesystem callbacks.
- FMOD allows file reading via disk, memory and TCP/IP. If you use your own file routines you will lose this ability.

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.

**See Also**

- [Channel::setPosition](#)
- **FMOD_CODEC_STATE**
- **FMOD_CODEC_DESCRIPTION**
- **FMOD_CODEC_OPENCALLBACK**
- **FMOD_CODEC_CLOSECALLBACK**
- **FMOD_CODEC_READCALLBACK**
- **FMOD_CODEC_GETLENGTHCALLBACK**
- **FMOD_CODEC_GETPOSITIONCALLBACK**
- **FMOD_CODEC_SOUNDCALLBACK**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
FMOD_CODEC_SOUNDCREATECALLBACK

Sound creation callback for the codec when FMOD finishes creating the sound. I.e. so the codec can set more parameters for the related created sound, i.e. loop points/mode or 3D attributes etc.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_CODEC_SOUNDCREATECALLBACK(
    FMOD_CODEC_STATE * codec_state,
    int subsound,
    FMOD_SOUND * sound
);
```

Parameters

`codec_state`

Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

`subsound`

Subsound index being created.

`soudn`

Pointer to the sound being created.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.

See Also

- [System::createSound](#)
- [System::createStream](#)
- **FMOD_CODEC_STATE**
- **FMOD_CODEC_DESCRIPTION**
- **FMOD_CODEC_OPENCALLBACK**
- **FMOD_CODEC_CLOSECALLBACK**
- **FMOD_CODEC_READCALLBACK**
- **FMOD_CODEC_GETLENGTHCALLBACK**
- **FMOD_CODEC_SETPOSITIONCALLBACK**
- **FMOD_CODEC_GETPOSITIONCALLBACK**
Firelight Technologies FMOD Ex
**FMOD_DSP_CREATECALLBACK**

This callback is called once when a user creates a DSP unit of this type. It is used to allocate memory, initialize variables and the like.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_CREATECALLBACK(
    FMOD_DSP_STATE * dsp_state
);
```

**Parameters**

*dsp_state*

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data. Do not cast this to FMOD_DSP! The handle to the user created DSP handle is stored within the `FMOD_DSP_STATE` structure.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Functions that the user would have to call for this callback to be called.

- `System::createDSP`
- `System::createDSPByType`
- `System::createDSPByPlugin`

Sometimes a user will re-use a DSP unit instead of releasing it and creating a new one, so it may be useful to implement `FMOD_DSP_RESETCALLBACK` to
reset any variables or buffers when the user calls it.

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`.

See Also

- `FMOD_DSP_STATE`
- `System::createDSP`
- `System::createDSPByType`
- `System::createDSPByPlugin`
- `FMOD_DSP_RESETCALLBACK`
Firelight Technologies FMOD Ex
FMOD_DSP_DIALOGCALLBACK

This callback is called when the user wants the plugin to display a configuration dialog box. This is not always necessary, so this can be left blank if wanted.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_DIALOGCALLBACK(
    FMOD_DSP_STATE * dsp_state,
    void * hwnd,
    int show
);
```

Parameters

dsp_state

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data. Do not cast this to FMOD_DSP! The handle to the user created DSP handle is stored within the FMOD_DSP_STATE structure.

hwnd

This is the target hwnd to display the dialog in. It must not pop up on this hwnd, it must actually be drawn within it.

show

1 = show the dialog, 0 = hide/remove the dialog.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Remarks

Functions that the user would have to call for this callback to be called. **DSP::showConfigDialog**.

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.

See Also

- **FMOD_DSP_STATE**
- **DSP::showConfigDialog**
Firelight Technologies FMOD Ex
**FMOD_DSP_GETPARAMCALLBACK**

This callback is called when the user wants to get an indexed parameter from a DSP unit.

**C/C++ Syntax**

```c
FMODRESULT F_CALLBACK FMOD_DSP_GETPARAMCALLBACK(
    FMOD_DSP_STATE * dsp_state,
    int index,
    float * value,
    char * valuestr
);
```

**Parameters**

`dsp_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data. Do not cast this to FMOD_DSP! The handle to the user created DSP handle is stored within the `FMOD_DSP_STATE` structure.

`index`

The index into the parameter list for the parameter the user wants to get.

`value`

Pointer to a floating point variable to receive the selected parameter value.

`valuestr`

A pointer to a string to receive the value of the selected parameter, but in text form. This might be useful to display words instead of numbers. For example "ON" or "OFF" instead of 1.0 and 0.0. The length of the buffer being passed in is always 16 bytes, so do not exceed this.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

Functions that the user would have to call for this callback to be called. **DSP::getParameter**. **FMOD_DSP_GETPARAMCALLBACK**.

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.

See Also

- **FMOD_DSP_STATE**
- **DSP::getParameter**
- **FMOD_DSP_SETPARAMCALLBACK**
Firelight Technologies FMOD Ex
This callback is called back regularly when the unit has been created, inserted to the DSP network, and set to active by the user.

This callback requires the user to fill the output pointer with data. Incoming data is provided and may be filtered on its way to the output pointer.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_READCALLBACK(
    FMOD_DSP_STATE * dsp_state,
    float * inbuffer,
    float * outbuffer,
    unsigned int length,
    int inchannels,
    int outchannels
);
```

Parameters

dsp_state

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data. Do not cast this to FMOD_DSP! The handle to the user created DSP handle is stored within the FMOD_DSP_STATE structure.

inbuffer

Pointer to incoming floating point -1.0 to +1.0 ranged data.

outbuffer

Pointer to outgoing floating point -1.0 to +1.0 ranged data. The dsp writer must write to this pointer else there will be silence.

length
The length of the incoming and outgoing buffer in samples. To get the length of the buffer in bytes, the user must multiply this number by the number of channels coming in (and out, they may be different) and then multiply by 4 for 1 float = 4 bytes.

\textit{inchannels}

The number of channels of interleaved PCM data in the inbuffer parameter. A mono signal coming in would be 1. A stereo signal coming in would be 2.

\textit{outchannels}

The number of channels of interleaved PCM data in the outbuffer parameter. A mono signal going out would be 1. A stereo signal going out would be 2.

**Return Values**

If the function succeeds then the return value is \texttt{FMOD\_OK}.
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

**Remarks**

Functions that the user would have to call for this callback to be called. \textit{None}.
This callback is called automatically and periodically when the DSP engine updates.
For a read update to be called it would have to be enabled, and this is done with \texttt{DSP::setActive}.
The range of -1 to 1 is a soft limit. In the case of the inbuffer it is not guaranteed to be in that range, and in the case of the outbuffer FMOD will accept values outside that range. However all values will be clamped to the range of -1 to 1 in the final mix.

Remember to return \texttt{FMOD\_OK} at the bottom of the function, or an appropriate error code from \texttt{FMOD\_RESULT}. 
See Also

- FMOD_DSP_STATE
- DSP::setActive
Firelight Technologies FMOD Ex
**FMOD_DSP_RELEASECALLBACK**

This callback is called when the user releases the DSP unit. It is used to free any resources allocated during the course of the lifetime of the DSP or perform any shut down code needed to clean up the DSP unit.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_RELEASECALLBACK(
    FMOD_DSP_STATE * dsp_state
);
```

**Parameters**

* dsp_state

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data. Do not cast this to FMOD_DSP! The handle to the user created DSP handle is stored within the FMOD_DSP_STATE structure.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

**Remarks**

Functions that the user would have to call for this callback to be called. **DSP::release**

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from FMOD_RESULT.
See Also

- FMOD_DSP_STATE
- DSP::release
Firelight Technologies FMOD Ex
FMOD_DSP_RESETCALLBACK

This callback function is called by DSP::reset to allow the effect to reset itself to a default state.

This is useful if an effect is for example still holding audio data for a sound that has stopped, and the unit wants to be relocated to a new sound. Resetting the unit would clear any buffers, put the effect back to its initial state, and get it ready for new sound data.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_RESETCALLBACK(
    FMOD_DSP_STATE * dsp_state
);
```

Parameters

`dsp_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data. Do not cast this to FMOD_DSP! The handle to the user created DSP handle is stored within the FMOD_DSP_STATE structure.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Functions that the user would have to call for this callback to be called. DSP::reset
Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`.

**See Also**

- `FMOD_DSP_STATE`
- `DSP::reset`
Firelight Technologies FMOD Ex
FMOD_DSP_SETPARAMCALLBACK

This callback is called when the user wants to set a parameter for a DSP unit.

C/C++ Syntax

```cpp
FMOD_RESULT F_CALLBACK FMOD_DSP_SETPARAMCALLBACK(
    FMOD_DSP_STATE * dsp_state,
    int index,
    float value
);
```

Parameters

dsp_state

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data. Do not cast this to FMOD_DSP! The handle to the user created DSP handle is stored within the FMOD_DSP_STATE structure.

index

The index into the parameter list for the parameter the user wants to set.

value

The value passed in by the user to set for the selected parameter.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Remarks

Functions that the user would have to call for this callback to be called. DSP::setParameter.

Range checking is not needed. FMOD will clamp the incoming value to the specified min/max.
Remember to return FMOD_OK at the bottom of the function, or an appropriate error code from FMOD_RESULT.

See Also

- FMOD_DSP_STATE
- DSP::setParameter
- FMOD_DSP_GETPARAMCALLBACK
Firelight Technologies FMOD Ex
FMOD_DSP_SETPOSITIONCALLBACK

Callback that is called when the user sets the position of a channel with Channel::setPosition.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_SETPOSITIONCALLBACK(
    FMOD_DSP_STATE * dsp_state,
    unsigned int position
);
```

Parameters

dsp_state

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data. Do not cast this to FMOD_DSP! The handle to the user created DSP handle is stored within the FMOD_DSP_STATE structure.

position

Position in channel stream to set to. Units are PCM samples (ie FMOD_TIMEUNIT_PCM).

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Functions that the user would have to call for this callback to be called.
**Channel::setPosition.**
If a DSP unit is attached to a channel and the user calls `Channel::setPosition` then this function will be called.
Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`.

**See Also**

- `FMOD_DSP_STATE`
- `Channel::setPosition`
Firelight Technologies FMOD Ex
FMOD_FILEASYNCCANCELCALLBACK

Callback to notify user that the resources associated with the file are about to be freed. Any asynchronous operations must be cancelled at this point before returning from the callback.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_ASYNCCANCELCALLBACK(
    void * handle,
    void * userdata
);
```

Parameters

*handle*

This is the handle returned from the open callback to use for your own file routines.

*userdata*

Data initialized in the FMOD_FILE_OPENCALLBACK.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

If asynchronous operations on this file are not cancelled before returning from this callback, then FMOD will free the memory associated with the file and the
user routine may read to an invalid/freed pointer - causing a crash.

See Also

- System::setFileSystem
- FMOD_FILE_OPENCALLBACK
- FMOD_FILE_CLOSECALLBACK
- FMOD_FILE_READCALLBACK
- FMOD_FILE_SEEKCALLBACK
- FMOD_FILEASYNCREADCALLBACK
- FMOD ASYNCREADINFO
Firelight Technologies FMOD Ex
**FMOD_FILE_ASYNCREADCALLBACK**

Callback for reading from a file asynchronously. Different to **FMOD_FILE_READCALLBACK** in that this function can be returned from immediately without supplying any data, which will simply cause FMOD to wait internally for data.

The user will fill out the writeable variables in the **FMODASYNCREADINFO** structure (in any thread), and when the **FMODASYNCREADINFO::result** field is set to something other than **FMOD_ERR_NOTREADY** then it will continue.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_ASYNCREADCALLBACK(
    FMODASYNCREADINFO * info,
    void * userdata
);
```

**Parameters**

- **info**
  
  Pointer to **FMODASYNCREADINFO** structure. Use this structure for your own system, to obtain the pointer to write to and number of bytes to read, as well as other information.

- **userdata**
  
  Userdata initialized in the **FMOD_FILE_OPENCALLBACK**.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMODRESULT** enumeration.
To return immediately, return **FMOD_OK**.

If during your code path, there was a fatal error, for example if you ran out of memory return **FMOD_ERR_MEMORY** and this will cause FMOD to stop what it was trying to do and return the error back to the caller.

**Remarks**

**NOTE:** If userasyncread is processed in the main thread, then it will hang the application, because FMOD will wait internally until data is ready, and the main thread process will not be able to supply the data. For this reason the user's file access should normally be from a separate thread.

**See Also**

- [System::setFileSystem](#)
- [FMOD_FILE_OPENCALLBACK](#)
- [FMOD_FILE_CLOSECALLBACK](#)
- [FMOD_FILE_READCALLBACK](#)
- [FMOD_FILE_SEEKCALLBACK](#)
- [FMOD_FILEASYNCCANCELCALLBACK](#)
- [FMODASYNCREADINFO](#)
Firelight Technologies FMOD Ex
**FMOD_FILE_CLOSECALLBACK**

Callback for closing a file.

### C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_CLOSECALLBACK(
    void * handle,
    void * userdata
);
```

#### Parameters

*handle*

This is the handle returned from the open callback to use for your own file routines.

*userdata*

Userdata initialized in the [FMOD_FILE_OPENCALLBACK](#).

#### Return Values

If the function succeeds then the return value is [FMOD_OK](#).
If the function fails then the return value will be one of the values defined in the [FMOD_RESULT](#) enumeration.

#### Remarks

Close any user created file handle and perform any cleanup necessary for the file here. If the callback is from `System::attachFileSystem`, then the return value is ignored.
See Also

- System::setFileSystem
- System::attachFileSystem
- FMOD_FILE_OPENCALLBACK
- FMOD_FILE_READCALLBACK
- FMOD_FILE_SEEKCALLBACK
Firelight Technologies FMOD Ex
**FMOD_FILE_OPENCALLBACK**

Callback for opening a file.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_OPENCALLBACK(
    const char * name,
    int unicode,
    unsigned int * filesize,
    void ** handle,
    void ** userdata
);
```

**Parameters**

*name*

This is the filename passed in by the user. You may treat this as you like.

*unicode*

Tells the callback if the string being passed in is a double byte unicode string or not. You may have to support this unless you know the target application will not support unicode.

*filesize*

The size of the file to be passed back to fmod, in bytes.

*handle*

This is to store a handle generated by the user. This will be the handle that gets passed into the other callbacks. Optional but may be needed.

*userdata*

This is to store userdata to be passed into the other callbacks. Optional.
Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

Return the appropriate error code such as **FMOD_ERR_FILE_NOTFOUND** if the file fails to open. If the callback is from **System::attachFileSystem**, then the return value is ignored.

See Also

- **System::setFileSystem**
- **System::attachFileSystem**
- **FMOD_FILE_CLOSECALLBACK**
- **FMOD_FILE_READCALLBACK**
- **FMOD_FILE_SEEKCALLBACK**
Firelight Technologies FMOD Ex
FMOD_FILE_READCALLBACK

Callback for reading from a file.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_READCALLBACK(
    void * handle,
    void * buffer,
    unsigned int sizebytes,
    unsigned int * bytesread,
    void * userdata
);
```

Parameters

`handle`

This is the handle you returned from the open callback to use for your own file routines.

`buffer`

The buffer to read your data into.

`sizebytes`

The number of bytes to read.

`bytesread`

The number of bytes successfully read.

`userdata`

Userdata initialized in the FMOD_FILE_OPENCALLBACK.

Return Values
If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

If the callback is from `System::attachFileSystem`, then the return value is ignored.

**See Also**

- `System::setFileSystem`
- `System::attachFileSystem`
- `FMOD_FILE_OPENCALLBACK`
- `FMOD_FILE_CLOSECALLBACK`
- `FMOD_FILE_SEEKCALLBACK`
Firelight Technologies FMOD Ex
FMOD_FILE_SEEKCALLBACK

Callback for seeking within a file.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_SEEKCALLBACK(
    void * handle,
    unsigned int pos,
    void * userdata
);
```

Parameters

`handle`

This is the handle returned from the open callback to use for your own file routines.

`pos`

This is the position or offset to seek to in the file in bytes.

`userdata`

Data initialized in the FMOD_FILE_OPENCALLBACK.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also
- System::setFileSystem
- FMOD_FILE_OPENCALLBACK
- FMOD_FILE_CLOSECALLBACK
- FMOD_FILE_READCALLBACK

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
**FMOD_MEMORY_ALLOCCALLBACK**

Callback to allocate a block of memory.

**C/C++ Syntax**

```c
void * F_CALLBACK FMOD_MEMORY_ALLOCCALLBACK(
    unsigned int  size,
    FMOD_MEMORY_TYPE  type,
    const char *  sourcestr
);
```

**Parameters**

* **size**
  Size in bytes of the memory block to be allocated and returned.

* **type**
  Type of memory allocation.

* **sourcestr**
  Only valid (not null) in logging versions of FMOD. Gives a string with the fmod source code filename and line number in it, for better resource tracking.

**Return Values**

On success, a pointer to the newly allocated block of memory is returned. On failure, NULL is returned.

**Remarks**

Returning an aligned pointer, of 16 byte alignment is recommended for speed
purposes.

See Also

- Memory_Initialize
- Memory_GetStats
- FMOD_MEMORY_REALLOCCALLBACK
- FMOD_MEMORY_FREECALLBACK
- FMOD_MEMORY_TYPE
Firelight Technologies FMOD Ex
FMOD_MEMORY_FREECALLBACK

Callback to free a block of memory.

C/C++ Syntax

```c
void F_CALLBACK FMOD_MEMORY_FREECALLBACK(
    void * ptr,
    FMOD_MEMORY_TYPE type,
    const char * sourcestr
);
```

Parameters

ptr

Pointer to a pre-existing block of memory to be freed.

type

Type of memory to be freed.

sourcestr

Only valid (not null) in logging versions of FMOD. Gives a string with the fmod source code filename and line number in it, for better resource tracking.

Return Values

void

See Also

- Memory_Initialize
- Memory_GetStats
- FMOD_MEMORY_ALLOC_CALLBACK
- FMOD_MEMORY_REALLOC_CALLBACK
- FMOD_MEMORY_TYPE

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
**FMOD_MEMORY_REALLOCCALLBACK**

Callback to re-allocate a block of memory to a different size.

**C/C++ Syntax**

```c
void * F_CALLBACK FMOD_MEMORY_REALLOCCALLBACK(
    void * ptr,
    unsigned int size,
    FMOD_MEMORY_TYPE type,
    const char * sourcestr
);
```

**Parameters**

`ptr`

Pointer to a block of memory to be resized. If this is NULL then a new block of memory is simply allocated.

`size`

Size of the memory to be reallocated. The original memory must be preserved.

`type`

Type of memory allocation.

`sourcestr`

Only valid (not null) in logging versions of FMOD. Gives a string with the fmod source code filename and line number in it, for better resource tracking.

**Return Values**

On success, a pointer to the newly re-allocated block of memory is returned. On failure, NULL is returned.
Remarks

Returning an aligned pointer, of 16 byte alignment is recommended for speed purposes.

See Also

- Memory_Initialize
- Memory_GetStats
- FMOD_MEMORY_ALLOCCALLBACK
- FMOD_MEMORY_FREECALLBACK
- FMOD_MEMORY_TYPE
Firelight Technologies FMOD Ex
FMOD_OUTPUT_CLOSECALLBACK

Shut down callback which is called when the user calls System::close or System::release. (System::release calls System::close internally)

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_CLOSECALLBACK(
    FMOD_OUTPUT_STATE * output_state
);
```

Parameters

`output_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Remember to return FMOD_OK at the bottom of the function, or an appropriate error code from FMOD_RESULT.

See Also

- System::release
- System::close
Firelight Technologies FMOD Ex
**FMOD_OUTPUT_GETDRIVERCAP**

Called when the user calls `System::getDriverCaps`.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_GETDRIVERCAPSCALLBACK(
    FMOD_OUTPUT_STATE * output_state,
    int id,
    FMOD_CAPS * caps
);
```

**Parameters**

`output_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

`id`

Index into the total number of outputs possible, provided by the `FMOD_OUTPUT_GETNUMDRIVERSCALLBACK` callback.

`caps`

Address of a variable to receive the caps available by this output device. See `FMOD_CAPS`. Fill this in.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`.

See Also

- `System::getDriverCaps`
- `System::getDriverInfo`
- `System::getNumDrivers`
- `FMOD_OUTPUT_GETNUMDRIVERSCALLBACK`
Firelight Technologies FMOD Ex
FMOD_OUTPUT_GETDRIVERNAMECALLBACK

Called when the user calls `System::getDriverInfo`.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_GETDRIVERNAMECALLBACK(
    FMOD_OUTPUT_STATE * output_state,
    int id,
    char * name,
    int namelen
);
```

Parameters

`output_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

`id`

Index into the total number of outputs possible, provided by the `FMOD_OUTPUT_GETNUMDRIVERSCALLBACK` callback.

`name`

Address of a variable to receive the driver name relevant to the index passed in. Fill this in.

`namelen`

Length of name buffer being passed in by the user.

Return Values

If the function succeeds then the return value is `FMOD_OK`. 
If the function fails then the return value will be one of the values defined in the
FMOD_RESULT enumeration.

Remarks

Remember to return FMOD_OK at the bottom of the function, or an appropriate
error code from FMOD_RESULT.

See Also

- System::getDriverInfo
- System::getNumDrivers
- FMOD_OUTPUT_GETNUMDRIVERSCALLBACK

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
FMOD_OUTPUT_GETHANDLECALLBACK

Called when the user calls System::getOutputHandle.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_GETHANDLECALLBACK(
    FMOD_OUTPUT_STATE * output_state,
    void ** handle
);
```

Parameters

`output_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

`handle`

Address of a variable to receieve the current plugin's output 'handle'. This is only if the plugin writer wants to allow the user access to the main handle behind the plugin (for example the file handle in a file writer plugin). The pointer type must be published to the user somehow, as is done in fmod.h.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.
**FMOD_OUTPUT_GETNUMDRIVECALLBACK**

Called when the user calls `System::getNumDrivers`.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_GETNUMDRIVECALLBACK(
    FMOD_OUTPUT_STATE * output_state,
    int * numdrivers
);
```

**Parameters**

`output_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

`numdrivers`

Address of a variable to receive the number of output drivers in your plugin.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`. Optional. FMOD will assume 0 if this is not specified.
See Also

- System::getNumDrivers
- System::getDriverInfo
- FMOD_OUTPUT_GETDRIVERNAMECALLBACK
Firelight Technologies FMOD Ex
Returns the current PCM offset or playback position for the output stream.
Called from the mixer thread, only when the 'polling' member of
\texttt{FMOD\_OUTPUT\_DESCRIPTION} is set to \texttt{true}.
The internal FMOD output thread calls this function periodically to determine
if it should ask for a block of audio data or not.

\textbf{C/C++ Syntax}

\begin{verbatim}
FMOD\_RESULT F\_CALLBACK FMOD\_OUTPUT\_GETPOSITIONCALLBACK(
    FMOD\_OUTPUT\_STATE * output_state,
    unsigned int * pcm
);
\end{verbatim}

\textbf{Parameters}

\textit{output\_state}

Pointer to the plugin state. The user can use this variable to access runtime
plugin specific variables and plugin writer user data.

\textit{pcm}

\textbf{Return Values}

If the function succeeds then the return value is \texttt{FMOD\_OK}.
If the function fails then the return value will be one of the values defined in the
\texttt{FMOD\_RESULT} enumeration.

\textbf{See Also}

- \texttt{FMOD\_OUTPUT\_DESCRIPTION}
- \texttt{FMOD\_OUTPUT\_LOCKCALLBACK}
- \texttt{FMOD\_OUTPUT\_UNLOCKCALLBACK}
Firelight Technologies FMOD Ex
Initialization callback which is called when the user calls `System::init`.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_INITCALLBACK(
    FMOD_OUTPUT_STATE * output_state,
    int selecteddriver,
    FMOD_INITFLAGS flags,
    int * outputrate,
    int outputchannels,
    FMOD_SOUND_FORMAT * outputformat,
    int dspbufferlength,
    int dsnumbuffers,
    void * extradriverdata
);
```

Parameters

`output_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

`selecteddriver`

This is the selected driver id that the user chose from calling `System::setDriver`.

`flags`

Initialization flags passed in by the user.

`outputrate`

Output rate selected by the user. If not possible, change the rate to the closest match.

`outputchannels`
Output channel count selected by the user. For example 1 = mono output. 2 = stereo output.

*outputformat*

Output format specified by the user. If not possible to support, return `FMOD_ERR_FORMAT`.

*dspbufferlength*

Size of the buffer fmod will mix to in one mix update. This value is in PCM samples.

*dspnumbuffers*

Number of buffers fmod will mix to in a circular fashion. Multiply this by dspbufferlength to get the total size of the output sound buffer to allocate.

*extradrivertdata*

Data passed in by the user specific to this driver. May be used for any purpose.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`.

**See Also**
• **FMOD RESULT**
• **System::init**
• **System::setDriver**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
FMOD_OUTPUT_LOCKCALLBACK

Called from the mixer thread, only when the 'polling' member of FMOD_OUTPUT_DESCRIPTION is set to true.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_LOCKCALLBACK(
    FMOD_OUTPUT_STATE * output_state,
    unsigned int  offset,
    unsigned int  length,
    void **  ptr1,
    void **  ptr2,
    unsigned int * len1,
    unsigned int * len2
);
```

Parameters

`output_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

`offset`

Offset in bytes to the position the caller wants to lock in the sample buffer.

`length`

Number of bytes the caller want to lock in the sample buffer.

`ptr1`

Address of a pointer that will point to the first part of the locked data.

`ptr2`

Address of a pointer that will point to the second part of the locked data. This will be null if the data locked hasn't wrapped at the end of the buffer.
len1

Length of data in bytes that was locked for ptr1

len2

Length of data in bytes that was locked for ptr2. This will be 0 if the data locked hasn't wrapped at the end of the buffer.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- **FMOD_OUTPUT_DESCRIPTION**
- **FMOD_OUTPUT_UNLOCKCALLBACK**
- **FMOD_OUTPUT_GETPOSITIONCALLBACK**
Firelight Technologies FMOD Ex
Called by the plugin, when the 'polling' member of 
FMOD_OUTPUT_DESCRIPTION is set to false. 
Use this function from your own driver irq/timer to read some data from 
FMOD's DSP engine. All of the resulting output caused by playing sounds and 
specifying effects by the user will be mixed here and written to the memory 
provided by the plugin writer.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_READFROMMIXER( 
    FMOD_OUTPUT_STATE * output_state, 
    void * buffer, 
    unsigned int length 
);
```

Parameters

output_state

Pointer to the plugin state. The user can use this variable to access runtime 
plugin specific variables and plugin writer user data.

buffer

Plugin-writer provided memory for the FMOD Ex mixer to write to.

length

Length of the buffer in samples.

Return Values

If the function succeeds then the return value is FMOD_OK. 
If the function fails then the return value will be one of the values defined in the
FMOD_RESULT enumeration.
Firelight Technologies FMOD Ex
Called from the mixer thread, only when the 'polling' member of
**FMOD_OUTPUT_DESCRIPTION** is set to true.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_UNLOCKCALLBACK(
    FMOD_OUTPUT_STATE * output_state,
    void * ptr1,
    void * ptr2,
    unsigned int len1,
    unsigned int len2
);
```

**Parameters**

*output_state*

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

*ptr1*

Pointer to the 1st locked portion of sample data, from Sound::lock.

*ptr2*

Pointer to the 2nd locked portion of sample data, from Sound::lock.

*len1*

Length of data in bytes that was locked for ptr1

*len2*

Length of data in bytes that was locked for ptr2. This will be 0 if the data locked hasn't wrapped at the end of the buffer.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

This function is normally called after data has been read/written to from Sound::lock. This function will do any post processing necessary and if needed, send it to sound ram.

See Also

- **FMOD_OUTPUT_DESCRIPTION**
- **FMOD_OUTPUT_LOCKCALLBACK**
- **FMOD_OUTPUT_GETPOSITIONCALLBACK**
Firelight Technologies FMOD Ex
FMOD_OUTPUT_UPDATECALLBACK

Called when the user calls System::update.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_UPDATECALLBACK(
    FMOD_OUTPUT_STATE * output_state
);
```

Parameters

`output_state`

Pointer to the plugin state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`.

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
**FMOD_SOUND_NONBLOCKCALLBACK**

Callback to be called when a sound has finished loading.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_SOUND_NONBLOCKCALLBACK(
    FMOD_SOUND * sound,
    FMOD_RESULT result
);
```

**Parameters**

*sound*

Pointer to the sound. C++ users see remarks.

*result*

Error code. **FMOD_OK** if sound was created successfully, or an error code otherwise.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.  
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

**C++ Users.** Cast **FMOD_SOUND** * to **FMOD::Sound** * inside the callback and use as normal.

Return code currently ignored.
See Also

- **System::createSound**
- **FMOD_CREATESOUNDEXINFO**
Firelight Technologies FMOD Ex
**FMOD_SOUND_PCMREADCALLBACK**

Used for 2 purposes.
One use is for user created sounds when **FMOD_OPENUSER** is specified when creating the sound.
The other use is to 'piggyback' on FMOD's read functions when opening a normal sound, therefore the callee can read (rip) or even write back new PCM data while FMOD is opening the sound.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_SOUND_PCMREADCALLBACK(
    FMOD_SOUND * sound,
    void * data,
    unsigned int datalen
);
```

**Parameters**

*sound*

Pointer to the sound. C++ users see remarks.

*data*

Pointer to raw PCM data that the user can either read or write to.

*datalen*

Length of the data in bytes.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

C++ Users. Cast `FMOD_SOUND *` to `FMOD::Sound *` inside the callback and use as normal.

The format of the sound can be retrieved with `Sound::getFormat` from this callback. This will allow the user to determine what type of pointer to use if they are not sure what format the sound is. If the callback is used for the purpose of 'piggybacking' normal FMOD sound loads, then you do not have to do anything at all, and it can be treated as purely informational. The return value is also ignored.

See Also

- `Sound::getFormat`
- `FMOD_SOUND_PCMSETPOSCALLBACK`
- `System::createSound`
- `System::createStream`
- `FMOD_CREATESSOUNDEXINFO`
Firelight Technologies FMOD Ex
FMOD_SOUND_PCMSETPOSCALLBACK

Callback for when the caller calls a seeking function such as Channel::setTime or Channel::setPosition.
If the sound is a user created sound, this can be used to seek within the user's resource.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_SOUND_PCMSETPOSCALLBACK(
    FMOD_SOUND * sound,
    int subsound,
    unsigned int position,
    FMOD_TIMEUNIT postype
);
```

Parameters

**sound**

Pointer to the sound. C++ users see remarks.

**subsound**

In a multi subsound type sound (ie fsb/dls/cdda), this will contain the index into the list of sounds.

**position**

Position to seek to that has been requested. This value will be of format FMOD_TIMEUNIT and must be parsed to determine what it is. Generally FMOD_TIMEUNIT_PCM will be the most common format.

**postype**

Position type that the user wanted to seek with. If the sound is a user create sound and the seek type is unsupported return FMOD_ERR_FORMAT.
**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

**C++ Users.** Cast **FMOD_SOUND** * to **FMOD::Sound** * inside the callback and use as normal.

If the callback is used for the purpose of 'piggybacking' normal FMOD sound loads, then you do not have to do anything at all, and it can be treated as purely informational. The return value is also ignored.

**See Also**

- **FMOD_SOUND_PCMREADCALLBACK**
- **System::createSound**
- **System::createStream**
- **FMOD_CREATESOUNDEXINFO**
Firelight Technologies FMOD Ex
**FMOD_SYSTEM_CALLBACK**

Callback for system events.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_SYSTEM_CALLBACK(
    FMOD_SYSTEM * system,
    FMOD_SYSTEM_CALLBACKTYPE type
);
```

**Parameters**

*system*

Pointer to a system handle. Note this could be null if
**FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED** is triggered from the EventSystem.

*type*

The type of callback. Refer to **FMOD_SYSTEM_CALLBACKTYPE**.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the
**FMOD_RESULT** enumeration.

**Remarks**

**C++ Users.** Cast **FMOD_SYSTEM** * to **FMOD::System** * inside the callback and use as normal.

'commanddata1' and 'commanddata2' meanings.
These 2 values are set by the callback depending on what is happening in the callback and the type of callback.

- **FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED**
  *commanddata1*: Always 0.
  *commanddata2*: Always 0.

- **FMOD_SYSTEM_CALLBACKTYPE_MEMORYALLOCATIONFAILED**
  *commanddata1*: A string (char*) which represents the file and line number of the allocation inside FMOD.
  *commanddata2*: The size (int) of the requested allocation.

- **FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED**
  *commanddata1*: The handle of the created thread. See notes below for thread handle types
  *commanddata2*: A string (char*) which represents the name of the thread.

- **FMOD_SYSTEM_CALLBACKTYPE_BADDSPCONNECTION**
  *commanddata1*: Pointer to a FMOD::DSP object that was the target of the DSP connection.
  *commanddata2*: Pointer to a FMOD::DSP object that was the source of the DSP connection.

- **FMOD_SYSTEM_CALLBACKTYPE_BADDSPELEVEL**
  *commanddata1*: Pointer to a FMOD::DSP object that was trying to exceed the DSP tree level maximum.
  *commanddata2*: 0.

**Note!** For **FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED**, the user must call `System::update` for the callback to trigger! See **FMOD_SYSTEM_CALLBACKTYPE** for details.

**Note!** For **FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED**, the handle that is returned (via commanddata1) is different on each platform. The types to cast to are as follows.
- iPhone, Linux, Mac : pthread_t
- PS3 : sys_ppu_thread_t
- PSP : PSPThreadWrapper. This is a custom struct you can define as typedef
struct PSPThreadWrapper { SceUID id; int (*func)(void *param); void *param; }

- Wii: OSThread
- Win32, Win64, Xbox360: HANDLE

Here is an example of a system callback.

```c
FMOD_RESULT F_CALLBACK systemcallback(FMOD_SYSTEM *system, FMOD_SYST
{
    FMOD::System *sys = (FMOD::System *)system;

    switch (type)
    {
    case FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED:
    {
        int numdrivers;

        printf("NOTE : FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED\n"
            sys->getNumDrivers(&numdrivers);

        printf("Numdevices = %d\n", numdrivers);
        break;
    }
    case FMOD_SYSTEM_CALLBACKTYPE_MEMORYALLOCATIONFAILED:
    {
        printf("ERROR : FMOD_SYSTEM_CALLBACKTYPE_MEMORYALLOCATION\n"
            printf("%s._n", commanddata1);
            printf("%d byte\n", commanddata2);
            break;
        }
    case FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED:
    {
        printf("NOTE : FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED or\n"
            printf("Thread ID = %d\n", (int)commanddata1);
            printf("Thread Name = %s\n", (char *)commanddata2);
            break;
        }
    case FMOD_SYSTEM_CALLBACKTYPE_BADDSPCONNECTION:
    {
        FMOD::DSP *source = (FMOD::DSP *)commanddata1;
```

```c
```
FMOD::DSP *dest = (FMOD::DSP *)commanddata2;

printf("ERROR : FMOD_SYSTEM_CALLBACKTYPE_BADDSPCONNECTI(
if (source)
{
    char name[256];
    source->getInfo(name, 0,0,0,0);
    printf("SOURCE = %s\n", name);
}
if (dest)
{
    char name[256];
    dest->getInfo(name, 0,0,0,0);
    printf("DEST = %s\n", name);
}
break;
}
case FMOD_SYSTEM_CALLBACKTYPE_BADDSPLEVEL:
{
    FMOD::DSP *source = (FMOD::DSP *)commanddata1;

    printf("ERROR : FMOD_SYSTEM_CALLBACKTYPE_BADDSPLEVEL occurred.
    if (source)
    {
        char name[256];
        source->getInfo(name, 0,0,0,0);
        printf("SOURCE = %s\n", name);
    }
    break;
    }
}

return FMOD_OK;

See Also

- System::setCallback
- FMOD_SYSTEM_CALLBACKTYPE
- System::update
Firelight Technologies FMOD Ex
Structures

FMOD_360_EXTRADRIVERDATA  FMOD_ADVANCEDSETTINGS
FMODASYNCREADINFO
FMOD_CDTOC
FMOD_CODEC_DESCRIPTION
FMOD_CODEC_STATE
FMOD_CODEC_WAVE_FORMAT
FMOD_CREATESOUNDINDEXINFO
FMOD_DSP_DESCRIPTION
FMOD_DSP_PARAMETERDESC
FMOD_DSP_STATE
FMOD_GUID
FMOD_MEMORY_USAGE_DETAILS
FMOD_OUTPUT_DESCRIPTION
FMOD_OUTPUT_STATE
FMOD_REVERB_CHANNEL_PROPERTIES
FMOD_REVERB_PROPERTIES
FMOD_TAG
FMOD_VECTOR
Firelight Technologies FMOD Ex
**FMOD_360_EXTRADRIVERDATA**

Use this structure with `System::init` to set which processor(s) FMOD will create its threads on. Pass this structure in as the "extradriverdata" parameter in `System::init`.

**Structure**

```c
typedef struct {
    FMOD_THREAD thread_mixer;
    FMOD_THREAD thread_stream;
    FMOD_THREAD thread_nonblocking;
    FMOD_THREAD thread_file;
    FMOD_THREAD thread_xaudio;
    FMOD_THREAD thread_geometry;
    FMOD_THREAD thread_recording;
    void * xaudio2instance;
} FMOD_360_EXTRADRIVERDATA;
```

**Members**

*thread_mixer*

[in] FMOD software mixer thread. Optional, ie 0 or `FMOD_THREAD_DEFAULT` = default behaviour

*thread_stream*

[in] FMOD stream thread. Optional, ie 0 or `FMOD_THREAD_DEFAULT` = default behaviour

*thread_nonblocking*

[in] FMOD thread for `FMOD_NONBLOCKING`. Optional, ie 0 or `FMOD_THREAD_DEFAULT` = default behaviour

*thread_file*

[in] FMOD file thread. Optional, ie 0 or `FMOD_THREAD_DEFAULT` = default behaviour
**thread_xaudio**

[in] Thread XAudio runs on. Optional, ie 0 or FMOD_THREAD_DEFAULT = default behaviour

**thread_geometry**

[in] FMOD geometry processing thread. Optional, ie 0 or FMOD_THREAD_DEFAULT = default behaviour

**thread_recording**

[in] FMOD recording thread. Optional, ie 0 or FMOD_THREAD_DEFAULT = default behaviour

**xaudio2instance**

[in] Instance of already initialized Xaudio2. Optional. 0 = let FMOD initialize XAudio2.

**See Also**

- [FMOD_THREAD](#)
- [System::init](#)
Firelight Technologies FMOD Ex
FMOD_ADVANCEDSETTINGS

Settings for advanced features like configuring memory and cpu usage for the FMOD_CREATECOMPRESSESAMPLE feature.

Structure

typedef struct {
    int cbsize;
    int maxMPEGcodecs;
    int maxADPCMcodecs;
    int maxXMAcodecs;
    int maxCELTcodecs;
    int maxVORBIScodecs;
    int maxPCMcodecs;
    int ASIONumChannels;
    char ** ASIOChannelList;
    FMOD_SPEAKER * ASIOSpeakerList;
    int max3DReverbDSPs;
    float HRTFMinAngle;
    float HRTFMaxAngle;
    float HRTFFreq;
    float vol0virtualvol;
    int eventqueuesize;
    unsigned int defaultDecodeBufferSize;
    char * debugLogFilename;
    unsigned short profileport;
    unsigned int geometryMaxFadeTime;
    unsigned int maxSpectrumWaveDataBuffers;
    unsigned int musicSystemCacheDelay;
    float distanceFilterCenterFreq;
    unsigned int stackSizeStream;
    unsigned int stackSizeNonBlocking;
    unsigned int stackSizeMixer;
} FMOD_ADVANCEDSETTINGS;

Members

cbsize

[w] Size of this structure. Use sizeof(FMOD_ADVANCEDSETTINGS) NOTE: This must be set before calling System::getAdvancedSettings!
maxMPEGcodecs

[r/w] Optional. Specify 0 to ignore. For use with FMOD_CREATECOMPRESSESAMPLE only. Mpeg codecs consume 21,684 bytes per instance and this number will determine how many mpeg channels can be played simultaneously. Default = 32.

maxADPCMcodecs

[r/w] Optional. Specify 0 to ignore. For use with FMOD_CREATECOMPRESSESAMPLE only. ADPCM codecs consume 2,136 bytes per instance and this number will determine how many ADPCM channels can be played simultaneously. Default = 32.

maxXMAcodecs

[r/w] Optional. Specify 0 to ignore. For use with FMOD_CREATECOMPRESSESAMPLE only. XMA codecs consume 14,836 bytes per instance and this number will determine how many XMA channels can be played simultaneously. Default = 32.

maxCELTcodecs

[r/w] Optional. Specify 0 to ignore. For use with FMOD_CREATECOMPRESSESAMPLE only. CELT codecs consume 11,500 bytes per instance and this number will determine how many CELT channels can be played simultaneously. Default = 32.

maxVORBIScodecs

[r/w] Optional. Specify 0 to ignore. For use with FMOD_CREATECOMPRESSESAMPLE only. Vorbis codecs consume 12,000 bytes per instance and this number will determine how many Vorbis channels can be played simultaneously. Default = 32.

maxPCMcodecs

[r/w] Optional. Specify 0 to ignore. For use with PS3 only. PCM codecs consume 12,672 bytes per instance and this number will determine how many streams and PCM voices can be played simultaneously. Default = 16.
**ASIONumChannels**

[r/w] Optional. Specify 0 to ignore. Number of channels available on the ASIO device.

**ASIOChannelList**

[r/w] Optional. Specify 0 to ignore. Pointer to an array of strings (number of entries defined by ASIONumChannels) with ASIO channel names.

**ASIOSpeakerList**

[r/w] Optional. Specify 0 to ignore. Pointer to a list of speakers that the ASIO channels map to. This can be called after `System::init` to remap ASIO output.

**max3DReverbDSPs**

[r/w] Optional. Specify 0 to ignore. The max number of 3d reverb DSP's in the system. (NOTE: CURRENTLY DISABLED / UNUSED)

**HRTFMinAngle**

[r/w] Optional. For use with FMOD_INIT_HRTF_LOWPASS. The angle range (0-360) of a 3D sound in relation to the listener, at which the HRTF function begins to have an effect. 0 = in front of the listener. 180 = from 90 degrees to the left of the listener to 90 degrees to the right. 360 = behind the listener. Default = 180.0.

**HRTFMaxAngle**

[r/w] Optional. For use with FMOD_INIT_HRTF_LOWPASS. The angle range (0-360) of a 3D sound in relation to the listener, at which the HRTF function has maximum effect. 0 = front of the listener. 180 = from 90 degrees to the left of the listener to 90 degrees to the right. 360 = behind the listener. Default = 360.0.

**HRTFFreq**

[r/w] Optional. Specify 0 to ignore. For use with FMOD_INIT_HRTF_LOWPASS. The cutoff frequency of the HRTF's lowpass filter function when at maximum effect. (i.e. at HRTFMaxAngle). Default =
vol0virtualvol

[r/w] Optional. Specify 0 to ignore. For use with FMOD_INIT_VOL0_BECOMES_VIRTUAL. If this flag is used, and the volume is 0.0, then the sound will become virtual. Use this value to raise the threshold to a different point where a sound goes virtual.

eventqueuesize

[r/w] Optional. Specify 0 to ignore. For use with FMOD Event system only. Specifies the number of slots available for simultaneous non blocking loads, across all threads. Default = 32.

defaultDecodeBufferSize

[r/w] Optional. Specify 0 to ignore. For streams. This determines the default size of the double buffer (in milliseconds) that a stream uses. Default = 400ms

dbLogFilename

[r/w] Optional. Specify 0 to ignore. Gives fmod's logging system a path/filename. Normally the log is placed in the same directory as the executable and called fmod.log. When using System::getAdvancedSettings, provide at least 256 bytes of memory to copy into.

profileport

[r/w] Optional. Specify 0 to ignore. For use with FMOD_INIT_ENABLE_PROFILE. Specify the port to listen on for connections by the profiler application.

geometryMaxFadeTime

[r/w] Optional. Specify 0 to ignore. The maximum time in milliseconds it takes for a channel to fade to the new level when its occlusion changes.

maxSpectrumWaveDataBuffers
[r/w] Optional. Specify 0 to ignore. Tells System::init to allocate a pool of wavedata/spectrum buffers to prevent memory fragmentation, any additional buffers will be allocated normally.

musicSystemCacheDelay

[r/w] Optional. Specify 0 to ignore. The delay the music system should allow for loading a sample from disk (in milliseconds). Default = 400 ms.

distanceFilterCenterFreq

[r/w] Optional. Specify 0 to ignore. For use with FMOD_INIT_DISTANCE_FILTERING. The default center frequency in Hz for the distance filtering effect. Default = 1500.0.

stackSizeStream

[r/w] Optional. Specify 0 to ignore. Specify the stack size for the FMOD Stream thread in bytes. Useful for custom codecs that use excess stack. Default 49,152 (48kb)

stackSizeNonBlocking

[r/w] Optional. Specify 0 to ignore. Specify the stack size for the FMOD_NONBLOCKING loading thread. Useful for custom codecs that use excess stack. Default 65,536 (64kb)

stackSizeMixer

[r/w] Optional. Specify 0 to ignore. Specify the stack size for the FMOD mixer thread. Useful for custom dsps that use excess stack. Default 49,152 (48kb)

Remarks

maxMPEGcodecs / maxADPCMcodelcs / maxXMAcodecs will determine the maximum cpu usage of playing realtime samples. Use this to lower potential excess cpu usage and also control memory usage.
maxPCMcodecs is for use with PS3 only. It will determine the maximum number of PCM voices that can be played at once. This includes streams of any format and all sounds created *without* the FMOD_CREATECOMPRESSEDSAMPLE flag. Memory will be allocated for codecs 'up front' (during System::init) if these values are specified as non zero. If any are zero, it allocates memory for the codec whenever a file of the type in question is loaded. So if maxMPEGcodecs is 0 for example, it will allocate memory for the mpeg codecs the first time an mp3 is loaded or an mp3 based .FSB file is loaded.

Due to inefficient encoding techniques on certain .wav based ADPCM files, FMOD can can need an extra 29720 bytes per codec. This means for lowest memory consumption. Use FSB as it uses an optimal/small ADPCM block size.

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value. Members marked with [w] mean the variable can be written to. The user can set the value. Members marked with [r/w] are either read or write depending on if you are using System::setAdvancedSettings (w) or System::getAdvancedSettings (r).

**See Also**

- System::setAdvancedSettings
- System::getAdvancedSettings
- System::init
- FMOD_MODE
Firelight Technologies FMOD Ex
**FMOD_ASYNCREADINFO**

Structure that is passed into [FMOD_FILE_ASYNCREADCALLBACK](#). Use the information in this structure to perform

**Structure**

```c
typedef struct {
    void * handle;
    unsigned int offset;
    unsigned int sizebytes;
    int priority;
    void * buffer;
    unsigned int bytesRead;
    FMOD_RESULT result;
    void * userdata;
} FMOD_ASYNCREADINFO;
```

**Members**

*handle*

[r] The file handle that was filled out in the open callback.

*offset*

[r] Seek position, make sure you read from this file offset.

*sizebytes*

[r] How many bytes requested for read.

*priority*

[r] 0 = low importance. 100 = extremely important (ie 'must read now or stuttering may occur')

*buffer*

[w] Buffer to read file data into.
bytesread

[w] Fill this in before setting result code to tell FMOD how many bytes were read.

result

[r/w] Result code, FMOD_OK tells the system it is ready to consume the data. Set this last! Default value = FMOD_ERR_NOTREADY.

userdata

[r] User data pointer.

Remarks

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value. Members marked with [w] mean the variable can be written to. The user can set the value.

Instructions: write to 'buffer', and 'bytesread' BEFORE setting 'result'. As soon as result is set, FMOD will asynchronously continue internally using the data provided in this structure.

Set 'result' to the result expected from a normal file read callback. If the read was successful, set it to FMOD_OK. If it read some data but hit the end of the file, set it to FMOD_ERR_FILE_EOF. If a bad error occurred, return FMOD_ERR_FILE_BAD. If a disk was ejected, return FMOD_ERR_FILE_DISKEJECTED.

See Also

- FMOD_FILEASYNCREADCALLBACK
- FMOD_FILEASYNCCANCELCALLBACK
Firelight Technologies FMOD Ex
FMOD_CDTOC

Structure describing a CD/DVD table of contents

Structure

typedef struct {
   int numtracks;
   int min[100];
   int sec[100];
   int frame[100];
} FMOD_CDTOC;

Members

numtracks

[r] The number of tracks on the CD

min

[r] The start offset of each track in minutes

sec

[r] The start offset of each track in seconds

frame

[r] The start offset of each track in frames

Remarks

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.
Members marked with [w] mean the variable can be written to. The user can set the value.
See Also

- Sound::getTag
Firelight Technologies FMOD Ex
When creating a codec, declare one of these and provide the relevant callbacks and name for FMOD to use when it opens and reads a file.

**Structure**

```c
typedef struct {
    const char * name;
    unsigned int version;
    int defaultasstream;
    FMOD_TIMEUNIT timeunits;
    FMOD_CODEC_OPENCALLBACK open;
    FMOD_CODEC_CLOSECALLBACK close;
    FMOD_CODEC_READCALLBACK read;
    FMOD_CODEC_GETLENGTHCALLBACK getlength;
    FMOD_CODEC_SETPOSITIONCALLBACK setposition;
    FMOD_CODEC_GETPOSITIONCALLBACK getposition;
    FMOD_CODEC_SOUNDCREATECALLBACK soundcreate;
    FMOD_CODEC_GETWAVEFORMAT getwaveformat;
} FMOD_CODEC_DESCRIPTION;
```

**Members**

- **name**
  
  [in] Name of the codec.

- **version**
  
  [in] Plugin writer's version number.

- **defaultasstream**
  
  [in] Tells FMOD to open the file as a stream when calling System::createSound, and not a static sample. Should normally be 0 (FALSE), because generally the user wants to decode the file into memory when using System::createSound. Mainly used for formats that decode for a very long time, or could use large amounts of memory when decoded. Usually sequenced formats such as mod/s3m/xm/it/midi fall into this category. It is mainly to stop users that don't
know what they're doing from getting FMOD_ERR_MEMORY returned from createSound when they should have in fact called System::createStream or used FMOD_CREATESTREAM in System::createSound.

_timeunits_

[in] When setposition codec is called, only these time formats will be passed to the codec. Use bitwise OR to accumulate different types.

_open_

[in] Open callback for the codec for when FMOD tries to open a sound using this codec.

_close_

[in] Close callback for the codec for when FMOD tries to close a sound using this codec.

_read_

[in] Read callback for the codec for when FMOD tries to read some data from the file to the destination format (specified in the open callback).

_getlength_

[in] Callback to return the length of the song in whatever format required when Sound::getLength is called.

_setposition_

[in] Seek callback for the codec for when FMOD tries to seek within the file with Channel::setPosition.

_getposition_

[in] Tell callback for the codec for when FMOD tries to get the current position within the with Channel::getPosition.

_soundcreate_
[in] Sound creation callback for the codec when FMOD finishes creating the sound. (So the codec can set more parameters for the related created sound, ie loop points/mode or 3D attributes etc).

**getwaveformat**

[in] Callback to tell FMOD about the waveform of a particular subsound. This is to save memory, rather than saving 1000 FMOD_CODEC_WAVEFORMAT structures in the codec, the codec might have a more optimal way of storing this information.

**Remarks**

Members marked with [in] mean the variable can be written to. The user can set the value.
Members marked with [out] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.

**See Also**

- [FMOD_CODEC_STATE](#)

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
**FMOD_CODEC_STATE**

Codec plugin structure that is passed into each callback.

Set these numsubsounds and waveform members when called in FMOD_CODEC_OPENCALLBACK to tell fmod what sort of sound to create.

The format, channels and frequency tell FMOD what sort of hardware buffer to create when you initialize your code. So if you wrote an MP3 codec that decoded to stereo 16bit integer PCM, you would specify FMOD_SOUND_FORMAT_PCM16, and channels would be equal to 2.

**Structure**

```c
typedef struct {
  int numsubsounds;
  FMOD_CODEC_WAVEFORMAT * waveform;
  void * plugindata;
  void * filehandle;
  unsigned int filesize;
  FMOD_FILE_READCALLBACK fileread;
  FMOD_FILE_SEEKCALLBACK fileseek;
  FMOD_CODEC_METADATACALLBACK metadata;
} FMOD_CODEC_STATE;
```

**Members**

**numsubsounds**

[in] Number of 'subsounds' in this sound. Anything other than 0 makes it a 'container' format (ie CDDA/DLS/FSB etc which contain 1 or more subsounds). For most normal, single sound codec such as WAV/AIFF/MP3, this should be 0 as they are not a container for subsounds, they are the sound by itself.

**waveformat**

[in] Pointer to an array of format structures containing information about each sample. Can be 0 or NULL if FMOD_CODEC_GETWAVEFORMAT callback is
preferred. The number of entries here must equal the number of subsounds defined in the subsound parameter. If numsubsounds = 0 then there should be 1 instance of this structure.

**plugindata**

[in] Plugin writer created data the codec author wants to attach to this object.

**filehandle**

[out] This will return an internal FMOD file handle to use with the callbacks provided.

**filesize**

[out] This will contain the size of the file in bytes.

**fileread**

[out] This will return a callable FMOD file function to use from codec.

**fileseek**

[out] This will return a callable FMOD file function to use from codec.

**metadata**

[out] This will return a callable FMOD metadata function to use from codec.

**Remarks**

Members marked with [in] mean the variable can be written to. The user can set the value. Members marked with [out] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.

An FMOD file might be from disk, memory or internet, however the file may be opened by the user.
'numsubsounds' should be 0 if the file is a normal single sound stream or sound. Examples of this would be .WAV, .WMA, .MP3, .AIFF.

'numsubsounds' should be 1+ if the file is a container format, and does not contain wav data itself. Examples of these types would be CDDA (multiple CD tracks), FSB (contains multiple sounds), DLS (contain instruments).

The arrays of format, channel, frequency, length and blockalign should point to arrays of information based on how many subsounds are in the format. If the number of subsounds is 0 then it should point to 1 of each attribute, the same as if the number of subsounds was 1. If subsounds was 100 for example, each pointer should point to an array of 100 of each attribute.

When a sound has 1 or more subsounds, you must play the individual sounds specified by first obtaining the subsound with `Sound::getSubSound`.

**See Also**

- `FMOD_SOUND_FORMAT`
- `FMOD_FILE_READCALLBACK`
- `FMOD_FILE_SEEKCALLBACK`
- `FMOD_CODEC_METADATACALLBACK`
- `Sound::getSubSound`
- `Sound::getNumSubSounds`
Firelight Technologies FMOD Ex
**FMOD_CODEC_WAVEFORMAT**

Set these values marked 'in' to tell fmod what sort of sound to create. The format, channels and frequency tell FMOD what sort of hardware buffer to create when you initialize your code. So if you wrote an MP3 codec that decoded to stereo 16bit integer PCM, you would specify FMOD_SOUND_FORMAT_PCM16, and channels would be equal to 2. Members marked as 'out' are set by fmod. Do not modify these. Simply specify 0 for these values when declaring the structure, FMOD will fill in the values for you after creation with the correct function pointers.

**Structure**

```c
typedef struct {
    char     name[256];
    FMOD_SOUND_FORMAT  format;
    int      channels;
    int      frequency;
    unsigned int  lengthbytes;
    unsigned int  lengthpcm;
    int       blockalign;
    int       loopstart;
    int       loopend;
    FMOD_MODE   mode;
    unsigned int  channelmask;
} FMOD_CODEC_WAVEFORMAT;
```

**Members**

*name*

[in] Name of sound.

*format*

[in] Format for (decompressed) codec output, ie FMOD_SOUND_FORMAT_PCM8, FMOD_SOUND_FORMAT_PCM16.

*channels*
[in] Number of channels used by codec, ie mono = 1, stereo = 2.

*frequency*

[in] Default frequency in hz of the codec, ie 44100.

*lengthbytes*

[in] Length in bytes of the source data.

*lengthpcm*

[in] Length in decompressed, PCM samples of the file, ie length in seconds * frequency. Used for Sound::getLength and for memory allocation of static decompressed sample data.

*blockalign*

[in] Blockalign in decompressed, PCM samples of the optimal decode chunk size for this format. The codec read callback will be called in multiples of this value.

*loopstart*

[in] Loopstart in decompressed, PCM samples of file.

*loopend*


*mode*

[in] Mode to determine whether the sound should by default load as looping, non looping, 2d or 3d.

*channelmask*

[in] Microsoft speaker channel mask, as defined for WAVEFORMATEXTENSIBLE and is found in ksmedia.h. Leave at 0 to play in natural speaker order.
Remarks

Members marked with [in] mean the variable can be written to. The user can set the value.
Members marked with [out] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.

An FMOD file might be from disk, memory or network, however the file may be opened by the user.

'numsubsounds' should be 0 if the file is a normal single sound stream or sound. Examples of this would be .WAV, .WMA, .MP3, .AIFF.
'numsubsounds' should be 1+ if the file is a container format, and does not contain wav data itself. Examples of these types would be CDDA (multiple CD tracks), FSB (contains multiple sounds), MIDI/MOD/S3M/XM/IT (contain instruments).
The arrays of format, channel, frequency, length and blockalign should point to arrays of information based on how many subsounds are in the format. If the number of subsounds is 0 then it should point to 1 of each attribute, the same as if the number of subsounds was 1. If subsounds was 100 for example, each pointer should point to an array of 100 of each attribute.
When a sound has 1 or more subsounds, you must play the individual sounds specified by first obtaining the subsound with `Sound::getSubSound`.

See Also

- `FMOD_SOUND_FORMAT`
- `FMOD_FILE_READCALLBACK`
- `FMOD_FILE_SEEKCALBACK`
- `FMOD_CODEC_METADATACALLBACK`
- `Sound::getSubSound`
- `Sound::getNumSubSounds`
Firelight Technologies FMOD Ex
FMOD_CREATEausesoundexinfo

Use this structure with System::createSound when more control is needed over loading.

The possible reasons to use this with System::createSound are:

- Loading a file from memory.
- Loading a file from within another larger (possibly wad/pak) file, by giving the loader an offset and length.
- To create a user created / non file based sound.
- To specify a starting subsound to seek to within a multi-sample sounds (ie FSB/DLS/SF2) when created as a stream.
- To specify which subsounds to load for multi-sample sounds (ie FSB/DLS/SF2) so that memory is saved and only a subset is actually loaded/read from disk.
- To specify 'piggyback' read and seek callbacks for capture of sound data as fmod reads and decodes it. Useful for ripping decoded PCM data from sounds as they are loaded / played.
- To specify a MIDI DLS/SF2 sample set file to load when opening a MIDI file. See below on what members to fill for each of the above types of sound you want to create.

Structure

typedef struct {
    int cbsize;
    unsigned int length;
    unsigned int fileoffset;
    int numchannels;
    int defaultfrequency;
    FMOD_Sound_format format;
    unsigned int decodemuffersize;
    int initialsoudsubsounnd;
    int numsubssounds;
    int * inclusionlist;
    int inclusionlistnum;
    FMOD_Sound_PCMreadcallback pcmreadcallback;
    FMOD_Sound_PCMsetposcallback pcmsetposcallback;
    FMOD_Sound_NONblockcallback nonblockcallback;
}
const char * dlsname;
const char * encryptionkey;
int maxpolyphony;
void * userdata;
FMOD_SOUND_TYPE suggestedsoundtype;
FMOD_FILE_OPENCALLBACK useropen;
FMOD_FILE_CLOSECALLBACK userclose;
FMOD_FILE_READCALLBACK userread;
FMOD_FILE SEEKCALLBACK userseek;
FMOD_FILEASYNCREADCALLBACK userasyncread;
FMOD_FILEASYNCCANCELCALLBACK userasynccancel;
FMOD_SPEAKERMAPTYPE speakermap;
FMOD_SOUNDGROUP * initialsoundgroup;
unsigned int initialseekposition;
FMOD_TIMEUNIT initialseekpostype;
int ignoresetfilesystem;
int cddaforceaspi;
unsigned int audioqueuepolicy;
unsigned int minmidigranularity;
int nonblockthreadid;
} FMOD_CREATENSOUNDEXINFO;

Members

cbsize

[w] Size of this structure. This is used so the structure can be expanded in the future and still work on older versions of FMOD Ex.

length

[w] Optional. Specify 0 to ignore. Size in bytes of file to load, or sound to create (in this case only if FMOD_OPENUSER is used). Required if loading from memory. If 0 is specified, then it will use the size of the file (unless loading from memory then an error will be returned).

fileoffset

[w] Optional. Specify 0 to ignore. Offset from start of the file to start loading from. This is useful for loading files from inside big data files.

numchannels

[w] Optional. Specify 0 to ignore. Number of channels in a sound mandatory if
FMOD_OPENUSER or FMOD_OPENRAW is used.

**defaultfrequency**

[w] Optional. Specify 0 to ignore. Default frequency of sound in a sound mandatory if FMOD_OPENUSER or FMOD_OPENRAW is used. Other formats use the frequency determined by the file format.

**format**

[w] Optional. Specify 0 or FMOD_SOUND_FORMAT_NONE to ignore. Format of the sound mandatory if FMOD_OPENUSER or FMOD_OPENRAW is used. Other formats use the format determined by the file format.

**decodebuffersize**

[w] Optional. Specify 0 to ignore. For streams. This determines the size of the double buffer (in PCM samples) that a stream uses. Use this for user created streams if you want to determine the size of the callback buffer passed to you. Specify 0 to use FMOD's default size which is currently equivalent to 400ms of the sound format created/loaded.

**initialsubsound**

[w] Optional. Specify 0 to ignore. In a multi-sample file format such as .FSB/.DLS/.SF2, specify the initial subsound to seek to, only if FMOD_CREATESTREAM is used.

**numsubsounds**

[w] Optional. Specify 0 to ignore or have no subsounds. In a sound created with FMOD_OPENUSER, specify the number of subsounds that are accessible with Sound::getSubSound. If not created with FMOD_OPENUSER, this will limit the number of subsounds loaded within a multi-subsound file. If using FSB, then if FMOD_CREATESOUNDEXINFO::inclusionlist is used, this will shuffle subsounds down so that there are not any gaps. It will mean that the indices of the sounds will be different.

**inclusionlist**
[w] Optional. Specify 0 to ignore. In a multi-sample format such as .FSB/.DLS/.SF2 it may be desirable to specify only a subset of sounds to be loaded out of the whole file. This is an array of subsound indices to load into memory when created.

**inclusionlistnum**

[w] Optional. Specify 0 to ignore. This is the number of integers contained within the inclusionlist array.

**pcmreadcallback**

[w] Optional. Specify 0 to ignore. Callback to 'piggyback' on FMOD's read functions and accept or even write PCM data while FMOD is opening the sound. Used for user sounds created with FMOD_OPENUSER or for capturing decoded data as FMOD reads it.

**pcmsetposcallback**

[w] Optional. Specify 0 to ignore. Callback for when the user calls a seeking function such as Channel::setTime or Channel::setPosition within a multi-sample sound, and for when it is opened.

**nonblockcallback**

[w] Optional. Specify 0 to ignore. Callback for successful completion, or error while loading a sound that used the FMOD_NONBLOCKING flag.

**dlsname**

[w] Optional. Specify 0 to ignore. Filename for a DLS or SF2 sample set when loading a MIDI file. If not specified, on Windows it will attempt to open /windows/system32/drivers/gm.dls or /windows/system32/drivers/etc/gm.dls, on Mac it will attempt to load /System/Library/Components/CoreAudio.component/Contents/Resources/gs_instruments.dls, otherwise the MIDI will fail to open. Current DLS support is for level 1 of the specification.

**encryptionkey**
Optional. Specify 0 to ignore. Key for encrypted FSB file. Without this key an encrypted FSB file will not load.

maxpolyphony

Optional. Specify 0 to ignore. For sequenced formats with dynamic channel allocation such as .MID and .IT, this specifies the maximum voice count allowed while playing. .IT defaults to 64. .MID defaults to 32.

userdata

Optional. Specify 0 to ignore. This is user data to be attached to the sound during creation. Access via Sound::getUserData. Note: This is not passed to FMOD_FILE_OPENCALLBACK, that is a different userdata that is file specific.

suggestedsoundtype

Optional. Specify 0 or FMOD_SOUND_TYPE_UNKNOWN to ignore. Instead of scanning all codec types, use this to speed up loading by making it jump straight to this codec.

useropen

Optional. Specify 0 to ignore. Callback for opening this file.

userclose

Optional. Specify 0 to ignore. Callback for closing this file.

userread

Optional. Specify 0 to ignore. Callback for reading from this file.

userseek

Optional. Specify 0 to ignore. Callback for seeking within this file.

userasyncread

Optional. Specify 0 to ignore. Callback for seeking within this file.
userasynccancel
[w] Optional. Specify 0 to ignore. Callback for seeking within this file.

speakermap
[w] Optional. Specify 0 to ignore. Use this to differ the way fmod maps multichannel sounds to speakers. See FMOD_SPEAKERMAPTYPE for more.

initialsoundgroup
[w] Optional. Specify 0 to ignore. Specify a sound group if required, to put sound in as it is created.

initialseekposition
[w] Optional. Specify 0 to ignore. For streams. Specify an initial position to seek the stream to.

initialseekpostype
[w] Optional. Specify 0 to ignore. For streams. Specify the time unit for the position set in initialseekposition.

ignoreresetfilesystem
[w] Optional. Specify 0 to ignore. Set to 1 to use fmod's built in file system. Ignores setFileSystem callbacks and also FMOD_CREATESOUNEXINFO file callbacks. Useful for specific cases where you don't want to use your own file system but want to use fmod's file system (ie net streaming).

cddaforceaspi
[w] Optional. Specify 0 to ignore. For CDDA sounds only - if non-zero use ASPI instead of NTSCSI to access the specified CD/DVD device.

audioqueuepolicy
[w] Optional. Specify 0 or FMOD_AUDIOQUEUE_CODECPOLICY_DEFAULT to ignore. Policy used to determine whether hardware or software is used for decoding, see
FMOD_AUDIOQUEUE_CODECPOLICY for options (iOS >= 3.0 required, otherwise only hardware is available)

**minmidigranularity**

[w] Optional. Specify 0 to ignore. Allows you to set a minimum desired MIDI mixer granularity. Values smaller than 512 give greater than default accuracy at the cost of more CPU and vice versa. Specify 0 for default (512 samples).

**nonblockthreadid**

[w] Optional. Specify 0 to ignore. Specifies a thread index to execute non blocking load on. Allows for up to 5 threads to be used for loading at once. This is to avoid one load blocking another. Maximum value = 4.

**Remarks**

This structure is optional! Specify 0 or NULL in `System::createSound` if you don't need it!

**Loading a file from memory.**

- Create the sound using the FMOD_OPENMEMORY flag.
- Mandatory. Specify 'length' for the size of the memory block in bytes.
- Other flags are optional.

**Loading a file from within another larger (possibly wad/pak) file, by giving the loader an offset and length.**
- Mandatory. Specify 'fileoffset' and 'length'.
- Other flags are optional.

**To create a user created / non file based sound.**

- Create the sound using the FMOD_OPENCUSER flag.
- Mandatory. Specify 'defaultfrequency, 'numchannels' and 'format'.
- Other flags are optional.

**To specify a starting subsound to seek to and flush with, within a multi-sample stream (ie FSB/DLS/SF2).**
• Mandatory. Specify 'initialsubsound'.

To specify which subsounds to load for multi-sample sounds (ie FSB/DLS/SF2) so that memory is saved and only a subset is actually loaded/read from disk.

• Mandatory. Specify 'inclusionlist' and 'inclusionlistnum'.

To specify 'piggyback' read and seek callbacks for capture of sound data as fmod reads and decodes it. Useful for ripping decoded PCM data from sounds as they are loaded / played.

• Mandatory. Specify 'pcmreadcallback' and 'pcmseekcallback'.

To specify a MIDI DLS/SF2 sample set file to load when opening a MIDI file.

• Mandatory. Specify 'dlsname'.

Setting the 'decodebuffersize' is for cpu intensive codecs that may be causing stuttering, not file intensive codecs (ie those from CD or netstreams) which are normally altered with System::setStreamBufferSize. As an example of cpu intensive codecs, an mp3 file will take more cpu to decode than a PCM wav file. If you have a stuttering effect, then it is using more cpu than the decode buffer playback rate can keep up with. Increasing the decode buffersize will most likely solve this problem.

FSB codec. If inclusionlist and numsounds are used together, this will trigger a special mode where subsounds are shuffled down to save memory. (useful for large FSB files where you only want to load 1 sound). There will be no gaps, ie no null subsounds. As an example, if there are 10,000 subsounds and there is an inclusionlist with only 1 entry, and numsounds = 1, then subsound 0 will be that entry, and there will only be the memory allocated for 1 subsound. Previously there would still be 10,000 subsound pointers and other associated codec entries allocated along with it multiplied by 10,000.

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.
Members marked with [w] mean the variable can be written to. The user can set
the value.

See Also

- [System::createSound](#)
- [System::setStreamBufferSize](#)
- [FMOD_MODE](#)
- [FMOD_SOUND_FORMAT](#)
- [FMOD_SOUND_TYPE](#)
- [FMOD_SPEAKERMAPTYPE](#)
Firelight Technologies FMOD Ex
FMOD_DSP_DESCRIPTION

When creating a DSP unit, declare one of these and provide the relevant callbacks and name for FMOD to use when it creates and uses a DSP unit of this type.

Structure

typedef struct {
    char name[32];
    unsigned int version;
    int channels;
    FMOD_DSP_CREATECALLBACK create;
    FMOD_DSP_RELEASECALLBACK release;
    FMOD_DSP_RESETCALLBACK reset;
    FMOD_DSP_READCALLBACK read;
    FMOD_DSP_SETPOSITIONCALLBACK setposition;
    int numparameters;
    FMOD_DSP_PARAMETERDESC * paramdesc;
    FMOD_DSP_SETPARAMCALLBACK setparameter;
    FMOD_DSP_GETPARAMCALLBACK getparameter;
    FMOD_DSP_DIALOGCALLBACK config;
    int configwidth;
    int configheight;
    void * userdata;
} FMOD_DSP_DESCRIPTION;

Members

name

[w] Name of the unit to be displayed in the network.

version

[w] Plugin writer's version number.

channels

[w] Number of channels. Use 0 to process whatever number of channels is currently in the network. >0 would be mostly used if the unit is a unit that only
generates sound.

create

[w] Create callback. This is called when DSP unit is created. Can be null.

release

[w] Release callback. This is called just before the unit is freed so the user can do any cleanup needed for the unit. Can be null.

reset

[w] Reset callback. This is called by the user to reset any history buffers that may need resetting for a filter, when it is to be used or re-used for the first time to its initial clean state. Use to avoid clicks or artifacts.

read

[w] Read callback. Processing is done here. Can be null.

setPosition

[w] Set position callback. This is called if the unit wants to update its position info but not process data, or reset a cursor position internally if it is reading data from a certain source. Can be null.

numParameters

[w] Number of parameters used in this filter. The user finds this with DSP::getNumParameters

paramDesc

[w] Variable number of parameter structures.

setParameter

[w] This is called when the user calls DSP::setParameter. Can be null.

getParameter
This is called when the user calls DSP::getParameter. Can be null.

`config`

This is called when the user calls DSP::showConfigDialog. Can be used to display a dialog to configure the filter. Can be null.

`configwidth`

Width of config dialog graphic if there is one. 0 otherwise.

`configheight`

Height of config dialog graphic if there is one. 0 otherwise.

`userdata`

Optional. Specify 0 to ignore. This is user data to be attached to the DSP unit during creation. Access via DSP::getUserData.

**Remarks**

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.
Members marked with [w] mean the variable can be written to. The user can set the value.

There are 2 different ways to change a parameter in this architecture. One is to use DSP::setParameter / DSP::getParameter. This is platform independant and is dynamic, so new unknown plugins can have their parameters enumerated and used.
The other is to use DSP::showConfigDialog. This is platform specific and requires a GUI, and will display a dialog box to configure the plugin.

**See Also**
- System::createDSP
- FMOD_DSP_STATE

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
**FMOD_DSP_PARAMETERDESC**

Structure to define a parameter for a DSP unit.

**Structure**

```c
typedef struct {
    float min;
    float max;
    float defaultval;
    char name[16];
    char label[16];
    const char * description;
} FMOD_DSP_PARAMETERDESC;
```

**Members**

`min`

[w] Minimum value of the parameter (ie 100.0).

`max`

[w] Maximum value of the parameter (ie 22050.0).

`defaultval`

[w] Default value of parameter.

`name`

[w] Name of the parameter to be displayed (ie "Cutoff frequency").

`label`

[w] Short string to be put next to value to denote the unit type (ie "hz").

`description`

[w] Description of the parameter to be displayed as a help item / tooltip for this
Remarks

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value. Members marked with [w] mean the variable can be written to. The user can set the value.

See Also

- System::createDSP
- DSP::setParameter
Firelight Technologies FMOD Ex
FMOD_DSP_STATE

DSP plugin structure that is passed into each callback.

Structure

typedef struct {
   FMOD_DSP * instance;
   void * plugindata;
   unsigned short speakermask;
} FMOD_DSP_STATE;

Members

instance

[r] Handle to the DSP hand the user created. Not to be modified. C++ users cast to FMOD::DSP to use.

plugindata

[w] Plugin writer created data the output author wants to attach to this object.

speakermask

[w] Specifies which speakers the DSP effect is active on

Remarks

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value. Members marked with [w] mean the variable can be written to. The user can set the value.

See Also
- **FMOD_DSP_DESCRIPTION**

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
FMOD_GUID

Structure describing a globally unique identifier.

Structure

typedef struct {
    unsigned int Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
} FMOD_GUID;

Members

Data1

Specifies the first 8 hexadecimal digits of the GUID

Data2

Specifies the first group of 4 hexadecimal digits.

Data3

Specifies the second group of 4 hexadecimal digits.

Data4

Array of 8 bytes. The first 2 bytes contain the third group of 4 hexadecimal digits. The remaining 6 bytes contain the final 12 hexadecimal digits.

See Also

- System::getDriverInfo
Firelight Technologies FMOD Ex
FMOD_MEMORY_USAGE_DETAILS

Structure to be filled with detailed memory usage information of an FMOD object

Structure

typedef struct {
    unsigned int other;
    unsigned int string;
    unsigned int system;
    unsigned int plugins;
    unsigned int output;
    unsigned int channel;
    unsigned int channelgroup;
    unsigned int codec;
    unsigned int file;
    unsigned int sound;
    unsigned int secondaryram;
    unsigned int soundgroup;
    unsigned int streambuffer;
    unsigned int dspconnection;
    unsigned int dsp;
    unsigned int dspcodec;
    unsigned int profile;
    unsigned int recordbuffer;
    unsigned int reverb;
    unsigned int reverbchannelprops;
    unsigned int geometry;
    unsigned int syncpoint;
    unsigned int eventsystem;
    unsigned int musicsystem;
    unsigned int fev;
    unsigned int memoryfsb;
    unsigned int eventproject;
    unsigned int eventgroupi;
    unsigned int soundbankclass;
    unsigned int soundbanklist;
    unsigned int streaminstance;
    unsigned int sounddefclass;
    unsigned int sounddefdefclass;
    unsigned int sounddefpool;
    unsigned int reverbddef;
    unsigned int eventreverb;
    unsigned int userproperty;
}
unsigned int eventinstance;
unsigned int eventinstance_complex;
unsigned int eventinstance_simple;
unsigned int eventinstance_layer;
unsigned int eventinstance_sound;
unsigned int eventenvelope;
unsigned int eventenvelopedef;
unsigned int eventparameter;
unsigned int eventcategory;
unsigned int eventenvelopepoint;
unsigned int eventinstancepool;
} FMOD_MEMORY_USAGE_DETAILS;

Members

other
[out] Memory not accounted for by other types

string
[out] String data

system
[out] System object and various internals

plugins
[out] Plugin objects and internals

output
[out] Output module object and internals

channel
[out] Channel related memory

channelgroup
[out] ChannelGroup objects and internals
codec
[out] Codecs allocated for streaming

file
[out] File buffers and structures

sound
[out] Sound objects and internals

secondaryram
[out] Sound data stored in secondary RAM

soundgroup
[out] SoundGroup objects and internals

streambuffer
[out] Stream buffer memory

dspconnection
[out] DSPConnection objects and internals

dsp
[out] DSP implementation objects

dspcodec
[out] Realtime file format decoding DSP objects

profile
[out] Profiler memory footprint.

recordbuffer
[out] Buffer used to store recorded data from microphone

reverb

[out] Reverb implementation objects

reverbchannelprops

[out] Reverb channel properties structs

geometry

[out] Geometry objects and internals

syncpoint

[out] Sync point memory.

eventsystem

[out] EventSystem and various internals

musicsystem

[out] MusicSystem and various internals

fev

[out] Definition of objects contained in all loaded projects e.g. events, groups, categories

memoryfsb

[out] Data loaded with preloadFSB

eventproject

[out] EventProject objects and internals

eventgroupi
[out] EventGroup objects and internals

*soundbankclass*

[out] Objects used to manage wave banks

*soundbanklist*

[out] Data used to manage lists of wave bank usage

*streaminstance*

[out] Stream objects and internals

*sounddefclass*

[out] Sound definition objects

*sounddefdefclass*

[out] Sound definition static data objects

*sounddefpool*

[out] Sound definition pool data

*reverbdef*

[out] Reverb definition objects

*eventreverb*

[out] Reverb objects

*userproperty*

[out] User property objects

*eventinstance*

[out] Event instance base objects
eventinstance_complex
[out] Complex event instance objects

eventinstance_simple
[out] Simple event instance objects

eventinstance_layer
[out] Event layer instance objects

eventinstance_sound
[out] Event sound instance objects

eventenvelope
[out] Event envelope objects

eventenvelopedef
[out] Event envelope definition objects

eventparameter
[out] Event parameter objects

eventcategory
[out] Event category objects

eventenvelopepoint
[out] Event envelope point objects

eventinstancepool
[out] Event instance pool memory
Remarks

Every public FMOD class has a getMemoryInfo function which can be used to get detailed information on what memory resources are associated with the object in question. On return from getMemoryInfo, each member of this structure will hold the amount of memory used for its type in bytes.

Members marked with [in] mean the user sets the value before passing it to the function.
Members marked with [out] mean FMOD sets the value to be used after the function exits.

See Also

- System::getMemoryInfo
- EventSystem::getMemoryInfo
- FMOD_MEMBITS
- FMOD_EVENT_MEMBITS
FMOD_OUTPUT_DESCRIPTION

When creating an output, declare one of these and provide the relevant callbacks and name for FMOD to use when it opens and reads a file of this type.

Structure

```c
typedef struct {
    const char * name;
    unsigned int version;
    int polling;
    FMOD_OUTPUT_GETNUMDRIVERSCALLBACK getnumdrivers;
    FMOD_OUTPUT_GETDRIVERNAMECALLBACK getdrivername;
    FMOD_OUTPUT_GETDRIVERCAPSCALLBACK getdrivercaps;
    FMOD_OUTPUT_INITCALLBACK init;
    FMOD_OUTPUT_CLOSECALLBACK close;
    FMOD_OUTPUT_UPDATECALLBACK update;
    FMOD_OUTPUT_GETHANDLECALLBACK gethandle;
    FMOD_OUTPUT_GETPOSITIONCALLBACK getposition;
    FMOD_OUTPUT_LOCKCALLBACK lock;
    FMOD_OUTPUT_UNLOCKCALLBACK unlock;
} FMOD_OUTPUT_DESCRIPTION;
```

Members

`name`

[in] Name of the output.

`version`

[in] Plugin writer's version number.

`polling`

[in] If TRUE (non zero), this tells FMOD to start a thread and call getposition / lock / unlock for feeding data. If 0, the output is probably callback based, so all the plugin needs to do is call readfrommixer to the appropriate pointer.

`getnumdrivers`
[in] For sound device enumeration. This callback is to give System::getNumDrivers something to return.

getdrivername

[in] For sound device enumeration. This callback is to give System::getDriverName something to return.

getdrivercaps

[in] For sound device enumeration. This callback is to give System::getDriverCaps something to return.

init

[in] Initialization function for the output device. This is called from System::init.

close

[in] Cleanup / close down function for the output device. This is called from System::close.

update

[in] Update function that is called once a frame by the user. This is called from System::update.

gethandle

[in] This is called from System::getOutputHandle. This is just to return a pointer to the internal system device object that the system may be using.

getposition

[in] This is called from the FMOD software mixer thread if 'polling' = true. This returns a position value in samples so that FMOD knows where and when to fill its buffer.

lock

[in] This is called from the FMOD software mixer thread if 'polling' = true. This
function provides a pointer to data that FMOD can write to when software mixing.

unlock

[in] This is called from the FMOD software mixer thread if 'polling' = true. This optional function accepts the data that has been mixed and copies it or does whatever it needs to before sending it to the hardware.

Remarks

Members marked with [in] mean the variable can be written to. The user can set the value. Members marked with [out] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.

See Also

- FMOD_OUTPUT_STATE
Firelight Technologies FMOD Ex
**FMOD_OUTPUT_STATE**

Output plugin structure that is passed into each callback.

**Structure**

```c
typedef struct {
    void * plugindata;
    FMOD_OUTPUT_READFROMMIXER readfrommixer;
} FMOD_OUTPUT_STATE;
```

**Members**

*plugindata*

[in] Plugin writer created data the output author wants to attach to this object.

*readfrommixer*

[out] Function to update mixer and write the result to the provided pointer. Used from callback based output only. Polling based output uses lock/unlock/getposition.

**Remarks**

Members marked with [in] mean the variable can be written to. The user can set the value. Members marked with [out] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.

**See Also**

- [FMOD_OUTPUT_DESCRIPTION](#)
Firelight Technologies FMOD Ex
FMOD_REVERB_CHANNELPROPERTIES

Structure defining the properties for a reverb source, related to a FMOD channel.

Note the default reverb properties are the same as the
FMOD_PRESET_GENERIC preset.
Note that integer values that typically range from -10,000 to 1000 are
represented in decibels, and are of a logarithmic scale, not linear, whereas float
values are typically linear.
PORTABILITY: Each member has the platform it supports in braces ie (win32/wii).

The numerical values listed below are the maximum, minimum and default
values for each variable respectively.

Structure

typedef struct {
    int Direct;
    int Room;
    unsigned int Flags;
    FMOD_DSP * ConnectionPoint;
} FMOD_REVERB_CHANNELPROPERTIES;

Members

Direct

[r/w] -10000 1000 0 Direct path level (SUPPORTED:SFX)

Room

[r/w] -10000 1000 0 Room effect level (SUPPORTED:SFX)

Flags

[r/w] FMOD_REVERB_CHANNELFLAGS - modifies the behavior of properties (SUPPORTED:SFX)
ConnectionPoint

[r/w] See remarks. DSP network location to connect reverb for this channel. (SUPPORTED:SFX).

Remarks

SUPPORTED next to each parameter means the platform the parameter can be set on. Some platforms support all parameters and some don't.
WII means Nintendo Wii hardware reverb (must use FMOD_HARDWARE).
PSP means Playstation Portable hardware reverb (must use FMOD_HARDWARE).
SFX means FMOD SFX software reverb. This works on any platform that uses FMOD_SOFTWARE for loading sounds.
--- means unsupported/deprecated. Will either be removed or supported by SFX in the future.

'ConnectionPoint' Parameter. This parameter is for the FMOD software reverb only (known as SFX in the list above).
By default the dsp network connection for a channel and its reverb is between the 'SFX Reverb' unit, and the channel's wavetable/resampler/dspcodec/oscillator unit (the unit below the channel DSP head). NULL can be used for this parameter to make it use this default behaviour.
This parameter allows the user to connect the SFX reverb to somewhere else internally, for example the channel DSP head, or a related channelgroup. The event system uses this so that it can have the output of an event going to the reverb, instead of just the output of the event's channels (thereby ignoring event effects/submixes etc).
Do not use if you are unaware of DSP network connection issues. Leave it at the default of NULL instead.

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value.
Members marked with [w] mean the variable can be written to. The user can set the value.
Members marked with [r/w] are either read or write depending on if you are using Channel::setReverbProperties (w) or Channel::getReverbProperties (r).
See Also

- [Channel::setReverbProperties](#)
- [Channel::getReverbProperties](#)
- [FMOD_REVERB_CHANNELFLAGS](#)
Firelight Technologies FMOD Ex
**FMOD_REVERB_PROPERTIES**

Structure defining a reverb environment.

**Structure**

```c
typedef struct {
    int    Instance;
    int    Environment;
    float  EnvDiffusion;
    int    Room;
    int    RoomHF;
    int    RoomLF;
    float  DecayTime;
    float  DecayHFRatio;
    float  DecayLFRatio;
    int    Reflections;
    float  ReflectionsDelay;
    int    Reverb;
    float  ReverbDelay;
    float  ModulationTime;
    float  ModulationDepth;
    float  HFReference;
    float  LFReference;
    float  Diffusion;
    float  Density;
    unsigned int  Flags;
} FMOD_REVERB_PROPERTIES;
```

**Members**

*Instance*

[w] 0 3 0 Environment Instance. (SUPPORTED:SFX(4 instances) and Wii (3 instances))

*Environment*

[r/w] -1 25 -1 Sets all listener properties. -1 = OFF. (SUPPORTED:SFX(-1 only)/PSP)
EnvDiffusion

\( [r/w] 0.0 \ 1.0 \ 1.0 \) Environment diffusion (SUPPORTED:WII)

Room

\( [r/w] -10000 \ 0 \ -1000 \) Room effect level (at mid frequencies)
(SUPPORTED:SFX/WII/PSP)

RoomHF

\( [r/w] -10000 \ 0 \ -100 \) Relative room effect level at high frequencies
(SUPPORTED:SFX)

RoomLF

\( [r/w] -10000 \ 0 \ 0 \) Relative room effect level at low frequencies
(SUPPORTED:SFX)

DecayTime

\( [r/w] 0.1 \ 20.0 \ 1.49 \) Reverberation decay time at mid frequencies
(SUPPORTED:SFX/WII)

DecayHFRatio

\( [r/w] 0.1 \ 2.0 \ 0.83 \) High-frequency to mid-frequency decay time ratio
(SUPPORTED:SFX)

DecayLFRatio

\( [r/w] 0.1 \ 2.0 \ 1.0 \) Low-frequency to mid-frequency decay time ratio
(SUPPORTED:---)

Reflections

\( [r/w] -10000 \ 1000 \ -2602 \) Early reflections level relative to room effect
(SUPPORTED:SFX/WII)

ReflectionsDelay
Initial reflection delay time (SUPPORTED:SFX)

Reverb

Late reverberation level relative to room effect (SUPPORTED:SFX)

ReverbDelay

Late reverberation delay time relative to initial reflection (SUPPORTED:SFX/WII)

ModulationTime

Modulation depth (SUPPORTED:WII)

Reference high frequency (hz) (SUPPORTED:SFX)

Reference low frequency (hz) (SUPPORTED:SFX)

Value that controls the echo density in the late reverberation decay. (SUPPORTED:SFX)

Value that controls the modal density in the late reverberation decay (SUPPORTED:SFX)

FMOD_REVERB_FLAGS - modifies the behavior of above properties
Remarks

Note the default reverb properties are the same as the
FMOD_PRESET_GENERIC preset.
Note that integer values that typically range from -10,000 to 1000 are
represented in decibels, and are of a logarithmic scale, not linear, whereas float
values are always linear.

The numerical values listed below are the maximum, minimum and default
values for each variable respectively.

**SUPPORTED** next to each parameter means the platform the parameter can be
set on. Some platforms support all parameters and some don't.
WII means Nintendo Wii hardware reverb (must use FMOD_HARDWARE).
PSP means Playstation Portable hardware reverb (must use
FMOD_HARDWARE).
SFX means FMOD SFX software reverb. This works on any platform that uses
FMOD_SOFTWARE for loading sounds.
--- means unsupported/deprecated. Will either be removed or supported by SFX
in the future.

Nintendo Wii Notes:
This structure supports only limited parameters, and maps them to the Wii
hardware reverb as follows.
DecayTime = 'time'
ReverbDelay = 'predelay'
ModulationDepth = 'damping'
Reflections = 'coloration'
EnvDiffusion = 'crosstalk'
Room = 'mix'

Members marked with [r] mean the variable is modified by FMOD and is for
reading purposes only. Do not change this value.
Members marked with [w] mean the variable can be written to. The user can set
the value.
Members marked with [r/w] are either read or write depending on if you are
using System::setReverbProperties (w) or System::getReverbProperties (r).

See Also

- System::setReverbProperties
- System::getReverbProperties
- FMOD_REVERB_PRESETS
- FMOD_REVERB_FLAGS

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
**FMOD_TAG**

Structure describing a piece of tag data.

**Structure**

```c
typedef struct {
    FMOD_TAGTYPE type;
    FMOD_TAGDATATYPE datatype;
    char * name;
    void * data;
    unsigned int datalen;
    FMOD_BOOL updated;
} FMOD_TAG;
```

**Members**

* **type**
  
  [r] The type of this tag.

* **datatype**
  
  [r] The type of data that this tag contains

* **name**
  
  [r] The name of this tag i.e. "TITLE", "ARTIST" etc.

* **data**
  
  [r] Pointer to the tag data - its format is determined by the datatype member

* **datalen**
  
  [r] Length of the data contained in this tag

* **updated**
  
  [r] True if this tag has been updated since last being accessed with
Sound::getTag

Remarks

Members marked with [r] mean the variable is modified by FMOD and is for reading purposes only. Do not change this value. Members marked with [w] mean the variable can be written to. The user can set the value.

See Also

- Sound::getTag
- FMOD_TAGTYPE
- FMOD_TAGDATATYPE
Firelight Technologies FMOD Ex
**FMOD_VECTOR**

Structure describing a point in 3D space.

**Structure**

```c
typedef struct {
    float  x;
    float  y;
    float  z;
} FMOD_VECTOR;
```

**Members**

*x*

X co-ordinate in 3D space.

*y*

Y co-ordinate in 3D space.

*z*

Z co-ordinate in 3D space.

**Remarks**

FMOD uses a left handed co-ordinate system by default. To use a right handed co-ordinate system specify `FMOD_INIT_3D_RIGHTHANDED` from `FMOD_INITFLAGS` in System::init.

**See Also**

- [System::set3DListenerAttributes](#)
- System::get3DListenerAttributes
- Channel::set3DAttributes
- Channel::get3DAttributes
- Channel::set3DCustomRolloff
- Channel::get3DCustomRolloff
- Sound::set3DCustomRolloff
- Sound::get3DCustomRolloff
- Geometry::addPolygon
- Geometry::setPolygonVertex
- Geometry::getPolygonVertex
- Geometry::setRotation
- Geometry::getRotation
- Geometry::setPosition
- Geometry::getPosition
- Geometry::setScale
- Geometry::getScale
- FMOD_INITFLAGS
Firelight Technologies FMOD Ex
Defines

FMOD_CAPS FMOD_DEBUGLEVEL
FMOD_INITFLAGS
FMOD_MEMBITS
FMOD_MEMORY_TYPE
FMOD_MODE
FMOD_REVERB_CHANNELFLAGS
FMOD_REVERB_FLAGS
FMOD_REVERB_PRESETS
FMOD_TIMEUNIT
Firelight Technologies FMOD Ex
FMOD_CAPS

Bit fields to use with System::getDriverCaps to determine the capabilities of a card / output device. It is important to check FMOD_CAPS_HARDWARE_EMULATED on windows machines, to then adjust System::setDSPBufferSize to (1024, 10) to compensate for the higher latency.

Definition

```c
#define FMOD_CAPS_NONE 0x00000000
#define FMOD_CAPS_HARDWARE 0x00000001
#define FMOD_CAPS_HARDWARE_EMULATED 0x00000002
#define FMOD_CAPS_OUTPUT_MULTICHANNEL 0x00000004
#define FMOD_CAPS_OUTPUT_FORMAT_PCM8 0x00000008
#define FMOD_CAPS_OUTPUT_FORMAT_PCM16 0x00000010
#define FMOD_CAPS_OUTPUT_FORMAT_PCM24 0x00000020
#define FMOD_CAPS_OUTPUT_FORMAT_PCM32 0x00000040
#define FMOD_CAPS_OUTPUT_FORMAT_PCMFLOAT 0x00000080
#define FMOD_CAPS_REVERB_LIMITED 0x00002000
#define FMOD_CAPS_LOOPBACK 0x00004000
```

Values

**FMOD_CAPS_NONE**

Device has no special capabilities.

**FMOD_CAPS_HARDWARE**

Device supports hardware mixing.

**FMOD_CAPS_HARDWARE_EMULATED**

User has device set to 'Hardware acceleration = off' in control panel, and now extra 200ms latency is incurred.

**FMOD_CAPS_OUTPUT_MULTICHANNEL**

Device can do multichannel output, ie greater than 2 channels.
FMOD_CAPS_OUTPUT_FORMAT_PCM8
Device can output to 8bit integer PCM.

FMOD_CAPS_OUTPUT_FORMAT_PCM16
Device can output to 16bit integer PCM.

FMOD_CAPS_OUTPUT_FORMAT_PCM24
Device can output to 24bit integer PCM.

FMOD_CAPS_OUTPUT_FORMAT_PCM32
Device can output to 32bit integer PCM.

FMOD_CAPS_OUTPUT_FORMAT_PCMFLOAT
Device can output to 32bit floating point PCM.

FMOD_CAPS_REVERB_LIMITED
Device supports some form of limited hardware reverb, maybe parameterless and only selectable by environment.

FMOD_CAPS_LOOPBACK
Device is a loopback recording device

See Also

- System::getDriverCaps
- System::setDSPBufferSize
Firelight Technologies FMOD Ex
**FMOD_DEBUGLEVEL**

Bit fields to use with FMOD::Debug_SetLevel / FMOD::Debug_GetLevel to control the level of tty debug output with logging versions of FMOD (fmodL).

**Definition**

```c
#define FMOD_DEBUG_LEVEL_NONE   0x00000000
#define FMOD_DEBUG_LEVEL_LOG    0x00000001
#define FMOD_DEBUG_LEVEL_ERROR  0x00000002
#define FMOD_DEBUG_LEVEL_WARNING 0x00000004
#define FMOD_DEBUG_LEVEL_HINT   0x00000008
#define FMOD_DEBUG_LEVEL_ALL    0x000000FF
#define FMOD_DEBUG_TYPE_MEMORY  0x00000100
#define FMOD_DEBUG_TYPE_THREAD  0x00000200
#define FMOD_DEBUG_TYPE_FILE    0x00000400
#define FMOD_DEBUG_TYPE_NET     0x00000800
#define FMOD_DEBUG_TYPE_EVENT   0x00001000
#define FMOD_DEBUG_TYPE_ALL     0x0000FFFF
#define FMOD_DEBUG_DISPLAY_TIMESTAMPS 0x01000000
#define FMOD_DEBUG_DISPLAY_LINENUMBERS 0x02000000
#define FMOD_DEBUG_DISPLAY_COMPRESS 0x04000000
#define FMOD_DEBUG_DISPLAY_THREAD 0x08000000
#define FMOD_DEBUG_DISPLAY_ALL  0x0F000000
#define FMOD_DEBUG_ALL          0xFFFFFFFF
```

**Values**

*FMOD_DEBUG_LEVEL_NONE*

Will display generic logging messages.

*FMOD_DEBUG_LEVEL_ERROR*

Will display errors.

*FMOD_DEBUG_LEVEL_WARNING*

Will display warnings that are not fatal.
**FMOD_DEBUG_LEVEL_HINT**

Will hint to you if there is something possibly better you could be doing.

**FMOD_DEBUG_LEVEL_ALL**

**FMOD_DEBUG_TYPE_MEMORY**

Show FMOD memory related logging messages.

**FMOD_DEBUG_TYPE_THREAD**

Show FMOD thread related logging messages.

**FMOD_DEBUG_TYPE_FILE**

Show FMOD file system related logging messages.

**FMOD_DEBUG_TYPE_NET**

Show FMOD network related logging messages.

**FMOD_DEBUG_TYPE_EVENT**

Show FMOD Event related logging messages.

**FMOD_DEBUG_TYPE_ALL**

**FMOD_DEBUG_DISPLAY_TIMESTAMPS**

Display the timestamp of the log entry in milliseconds.

**FMOD_DEBUG_DISPLAY_LINENUMBERS**

Display the FMOD Ex source code line numbers, for debugging purposes.

**FMOD_DEBUG_DISPLAY_COMPRESS**

If a message is repeated more than 5 times it will stop displaying it and instead display the number of times the message was logged.
**FMOD_DEBUG_DISPLAY_THREAD**

Display the thread ID of the calling function that caused this log entry to appear.

**FMOD_DEBUG_DISPLAY_ALL**

**FMOD_DEBUG_ALL**

**See Also**

- [Debug_SetLevel](#)
- [Debug_GetLevel](#)
Firelight Technologies FMOD Ex
**FMOD_INIT_FLAGS**

Initialization flags. Use them with System::init in the flags parameter to change various behavior. Use System::setAdvancedSettings to adjust settings for some of the features that are enabled by these flags.

**Definition**

```cpp
#define FMOD_INIT_NORMAL 0x00000000
#define FMOD_INIT_STREAM_FROM_UPDATE 0x00000001
#define FMOD_INIT_3D_RIGHTHANDED 0x00000002
#define FMOD_INIT_SOFTWARE_DISABLE 0x00000004
#define FMOD_INIT_OCCLUSION_LOWPASS 0x00000008
#define FMOD_INIT_HRTF_LOWPASS 0x00000010
#define FMOD_INIT_DISTANCE_FILTERING 0x00000020
#define FMOD_INIT_SOFTWARE_REVERB_LOWMEM 0x00000040
#define FMOD_INIT_ENABLE_PROFILE 0x00000020
#define FMOD_INIT_VOL0_BECOMES_VIRTUAL 0x00000080
#define FMOD_INIT_WASAPI_EXCLUSIVE 0x00000100
#define FMOD_INIT_PS3_PREFERDTS 0x00000200
#define FMOD_INIT_PS3_FORCE2CHLPCM 0x00000400
#define FMOD_INIT_DISABLE_MYEARS_AUTODETECT 0x00000800
#define FMOD_INIT_SYSTEM_MUSICMUTENOTPAUSE 0x00001000
#define FMOD_INIT_SYNCMIXERWITHUPDATE 0x00002000
#define FMOD_INIT_GEOMETRY_USECLOSEST 0x00004000
#define FMOD_INIT_DISABLE_MYEARS_AUTODETECT 0x00008000
#define FMOD_INIT_PS3_DISABLE_DTS 0x00000100
#define FMOD_INIT_PS3_DISABLE_DOLBY_DIGITAL 0x00000200
```

**Values**

*FMOD_INIT_NORMAL*

All platforms - Initialize normally

*FMOD_INIT_STREAM_FROM_UPDATE*

All platforms - No stream thread is created internally. Streams are driven from System::update. Mainly used with non-realtime outputs.

*FMOD_INIT_3D_RIGHTHANDED*
All platforms - FMOD will treat +X as right, +Y as up and +Z as backwards (towards you).

**FMOD_INIT_SOFTWARE_DISABLE**

All platforms - Disable software mixer to save memory. Anything created with FMOD_SOFTWARE will fail and DSP will not work.

**FMOD_INIT_OCCLUSION_LOWPASS**

All platforms - All FMOD_SOFTWARE (and FMOD_HARDWARE on 3DS and NGP) with FMOD_3D based voices will add a software lowpass filter effect into the DSP chain which is automatically used when `Channel::set3DOcclusion` is used or the geometry API.

**FMOD_INIT_HRTF_LOWPASS**

All platforms - All FMOD_SOFTWARE (and FMOD_HARDWARE on 3DS and NGP) with FMOD_3D based voices will add a software lowpass filter effect into the DSP chain which causes sounds to sound duller when the sound goes behind the listener. Use `System::setAdvancedSettings` to adjust cutoff frequency.

**FMOD_INIT_DISTANCE_FILTERING**

All platforms - All FMOD_SOFTWARE with FMOD_3D based voices will add a software lowpass and highpass filter effect into the DSP chain which will act as a distance-automated bandpass filter. Use `System::setAdvancedSettings` to adjust the center frequency.

**FMOD_INIT_SOFTWARE_REVERB_LOWMEM**

All platforms - SFX reverb is run using 22/24khz delay buffers, halving the memory required.

**FMOD_INIT_ENABLE_PROFILE**

All platforms - Enable TCP/IP based host which allows FMOD Designer or FMOD Profiler to connect to it, and view memory, CPU and the DSP network graph in real-time.
**FMOD_INIT_VOL0_BECOMES_VIRTUAL**

All platforms - Any sounds that are 0 volume will go virtual and not be processed except for having their positions updated virtually. Use `System::setAdvancedSettings` to adjust what volume besides zero to switch to virtual at.

**FMOD_INIT_WASAPI_EXCLUSIVE**

Win32 Vista only - for WASAPI output - Enable exclusive access to hardware, lower latency at the expense of excluding other applications from accessing the audio hardware.

**FMOD_INIT_PS3_PREFERDTS**

PS3 only - Prefer DTS over Dolby Digital if both are supported. Note: 8 and 6 channel LPCM is always preferred over both DTS and Dolby Digital.

**FMOD_INIT_PS3_FORCE2CHLPCM**

PS3 only - Force PS3 system output mode to 2 channel LPCM.

**FMOD_INIT_DISABLEDOLBY**

Wii / 3DS - Disable Dolby Pro Logic surround. Speakermode will be set to STEREO even if user has selected surround in the system settings.

**FMOD_INIT_SYSTEM_MUSICMUTENOTPAUSE**

Xbox 360 / PS3 - The "music" channelgroup which by default pauses when custom 360 dashboard / PS3 BGM music is played, can be changed to mute (therefore continues playing) instead of pausing, by using this flag.

**FMOD_INIT_SYNCMIXERWITHUPDATE**

Win32/Wii/PS3/Xbox/Xbox 360 - FMOD Mixer thread is woken up to do a mix when `System::update` is called rather than waking periodically on its own timer.

**FMOD_INIT_GEOMETRY_USECLOSEST**

All platforms - With the geometry engine, only process the closest polygon
rather than accumulating all polygons the sound to listener line intersects.

**FMOD_INIT_DISABLE_MYEARS_AUTODETECT**

Win32 - Disables automatic setting of FMOD_SPEAKERMODE_STEREO to FMOD_SPEAKERMODE_MYEARS if the MyEars profile exists on the PC. MyEars is HRTF 7.1 downmixing through headphones.

**FMOD_INIT_PS3_DISABLEDTS**

PS3 only - Disable DTS output mode selection

**FMOD_INIT_PS3_DISABLEDDOLBYDIGITAL**

PS3 only - Disable Dolby Digital output mode selection

**See Also**

- System::init
- System::update
- System::setAdvancedSettings
- Channel::set3DOcclusion
Firelight Technologies FMOD Ex
**FMOD_MEMBITS**

Bitfield used to request specific memory usage information from the getMemoryInfo function of every public FMOD Ex class. Use with the "memorybits" parameter of getMemoryInfo to get information on FMOD Ex memory usage. Every public FMOD class has a getMemoryInfo function which can be used to get detailed information on what memory resources are associated with the object in question. The FMOD_MEMBITS defines can be OR'd together to specify precisely what memory usage you'd like to get information on. See `System::getMemoryInfo` for an example.

**Definition**

```c
#define FMOD_MEMBITS_OTHER 0x00000001
#define FMOD_MEMBITS_STRING 0x00000002
#define FMOD_MEMBITS_SYSTEM 0x00000004
#define FMOD_MEMBITS_PLUGINS 0x00000008
#define FMOD_MEMBITS_OUTPUT 0x00000010
#define FMOD_MEMBITS_CHANNEL 0x00000020
#define FMOD_MEMBITS_CHANNELGROUP 0x00000040
#define FMOD_MEMBITS_CODEC 0x00000080
#define FMOD_MEMBITS_FILE 0x00000100
#define FMOD_MEMBITS_SOUND 0x00000200
#define FMOD_MEMBITS_SOUND_SECONDARYRAM 0x00000400
#define FMOD_MEMBITS_SOUNDGROUP 0x00000800
#define FMOD_MEMBITS_STREAMBUFFER 0x00001000
#define FMOD_MEMBITS_DSPCONNECTION 0x00002000
#define FMOD_MEMBITS_DSP 0x00004000
#define FMOD_MEMBITS_DSPCODEC 0x00008000
#define FMOD_MEMBITS_PROFILE 0x00010000
#define FMOD_MEMBITS_RECORDBUFFER 0x00020000
#define FMOD_MEMBITS_REVERB 0x00040000
#define FMOD_MEMBITS_REVERBCHANNELPROPS 0x00080000
#define FMOD_MEMBITS_GEOMETRY 0x00100000
#define FMOD_MEMBITS_SYNCPOINT 0x00200000
#define FMOD_MEMBITS_ALL 0xffffffff
```

**Values**

**FMOD_MEMBITS_OTHER**

Memory not accounted for by other types
FMOD_MEMBITS_STRING
String data

FMOD_MEMBITS_SYSTEM
System object and various internals

FMOD_MEMBITS_PLUGIN
Plugin objects and internals

FMOD_MEMBITS_OUTPUT
Output module object and internals

FMOD_MEMBITS_CHANNEL
Channel related memory

FMOD_MEMBITS_CHANNELGROUP
ChannelGroup objects and internals

FMOD_MEMBITS_CODEC
Codecs allocated for streaming

FMOD_MEMBITS_FILE
Codecs allocated for streaming

FMOD_MEMBITS_SOUND
Sound objects and internals

FMOD_MEMBITS_SOUND_SECONDARYRAM
Sound data stored in secondary RAM

FMOD_MEMBITS_SOUNDGROUP
SoundGroup objects and internals

*FMOD_MEMBITS_STREAMBUFFER*
Stream buffer memory

*FMOD_MEMBITS_DSPCONNECTION*
DSPConnection objects and internals

*FMOD_MEMBITS_DSP*
DSP implementation objects

*FMOD_MEMBITS_DSPCODEC*
Realtime file format decoding DSP objects

*FMOD_MEMBITS_PROFILE*
Profiler memory footprint.

*FMOD_MEMBITS_RECORDBUFFER*
Buffer used to store recorded data from microphone

*FMOD_MEMBITS_REVERB*
Reverb implementation objects

*FMOD_MEMBITS_REVERBCHANNELPROPS*
Reverb channel properties structs

*FMOD_MEMBITS_GEOMETRY*
Geometry objects and internals

*FMOD_MEMBITS_SYNCPOINT*
Sync point memory.
**FMOD_MEMBITS_ALL**

All memory used by FMOD Ex

**See Also**

- [FMOD_EVENT_MEMBITS](#)
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
FMOD_MEMORY_TYPE

Bit fields for memory allocation type being passed into FMOD memory callbacks. Remember this is a bitfield. You may get more than 1 bit set (ie physical + persistent) so do not simply switch on the types! You must check each bit individually or clear out the bits that you do not want within the callback. Bits can be excluded if you want during Memory_Initialize so that you never get them.

Definition

```c
#define FMOD_MEMORY_NORMAL   0x00000000
#define FMOD_MEMORY_STREAM_FILE  0x00000001
#define FMOD_MEMORY_STREAM_DECODE  0x00000002
#define FMOD_MEMORY_SAMPLEDATA    0x00000004
#define FMOD_MEMORY_DSP_OUTPUTBUFFER 0x00000008
#define FMOD_MEMORY_XBOX360_PHYSICAL 0x00100000
#define FMOD_MEMORY_PERSISTENT    0x00200000
#define FMOD_MEMORY_SECONDARY    0x00400000
#define FMOD_MEMORY_ALL          0xFFFFFFFF
```

Values

**FMOD_MEMORY_NORMAL**

Standard memory.

**FMOD_MEMORY_STREAM_FILE**

Stream file buffer, size controllable with System::setStreamBufferSize.

**FMOD_MEMORY_STREAM_DECODE**

Stream decode buffer, size controllable with FMOD_CREATE SOUNDEXINFO::decodebuffersize.

**FMOD_MEMORY_SAMPLEDATA**

Sample data buffer. Raw audio data, usually PCM/MPEG/ADPCM/XMA data.
**FMOD_MEMORY_DSP_OUTPUTBUFFER**

DSP memory block allocated when more than 1 output exists on a DSP node.

**FMOD_MEMORY_XBOX360_PHYSICAL**

Requires XPhysicalAlloc / XPhysicalFree.

**FMOD_MEMORY_PERSISTENT**

Persistent memory. Memory will be freed when System::release is called.

**FMOD_MEMORY_SECONDARY**

Secondary memory. Allocation should be in secondary memory. For example RSX on the PS3.

**FMOD_MEMORY_ALL**


**See Also**

- [FMOD_MEMORY_ALLOCCALLBACK](#)
- [FMOD_MEMORY_REALLOCCALLBACK](#)
- [FMOD_MEMORY_FREECALLBACK](#)
- [Memory_Initialize](#)
Firelight Technologies FMOD Ex
FMOD_MODE

Sound description bitfields, bitwise OR them together for loading and describing sounds. By default a sound will open as a static sound that is decompressed fully into memory to PCM. (ie equivalent of FMOD_CREATESAMPLE)

To have a sound stream instead, use FMOD_CREATESTREAM, or use the wrapper function `System::createStream`.

Some opening modes (ie FMOD_OPENUSER, FMOD_OPENMEMORY, FMOD_OPENMEMORY_POINT, FMOD_OPENNRA) will need extra information.

This can be provided using the FMOD_CREATESOUNDEXINFO structure. Specifying FMOD_OPENMEMORY_POINT will POINT to your memory rather allocating its own sound buffers and duplicating it internally.

This means you cannot free the memory while FMOD is using it, until after `Sound::release` is called. With FMOD_OPENMEMORY_POINT, for PCM formats, only WAV, FSB, and RAW are supported. For compressed formats, only those formats supported by FMOD_CREATECOMPRESSEDSAMPLE are supported.

With FMOD_OPENMEMORY_POINT and FMOD_OPENNRA or PCM, if using them together, note that you must pad the data on each side by 16 bytes. This is so fmod can modify the ends of the data for looping/interpolation/mixing purposes. If a wav file, you will need to insert silence, and then reset loop points to stop the playback from playing that silence.

With FMOD_OPENMEMORY_POINT, For Wii/PSP FMOD_HARDWARE supports this flag for the GCADPCM/VAG formats. On other platforms FMOD_SOFTWARE must be used.

Xbox 360 memory On Xbox 360 Specifying FMOD_OPENMEMORY_POINT to a virtual memory address will cause FMOD_ERR_INVALID_ADDRESS to be returned. Use physical memory only for this functionality.

FMOD_LOWMEM is used on a sound if you want to minimize the memory overhead, by having FMOD not allocate memory for certain features that are not likely to be used in a game environment. These are:

1. Sound::getName functionality is removed. 256 bytes per sound is saved.
Definition

#define FMOD_DEFAULT 0x00000000
#define FMOD_LOOP_OFF 0x00000001
#define FMOD_LOOP_NORMAL 0x00000002
#define FMOD_LOOP_BIDI 0x00000004
#define FMOD_2D 0x00000008
#define FMOD_3D 0x00000010
#define FMOD_HARDWARE 0x00000020
#define FMOD_SOFTWARE 0x00000040
#define FMOD_CREATESTREAM 0x00000080
#define FMOD_CREATESAMPLE 0x00000020
#define FMOD_OPENUSER 0x00000400
#define FMOD_OPENMEMORY 0x00000800
#define FMOD_OPENMEMORY_POINT 0x10000000
#define FMOD_OPENRAW 0x00001000
#define FMOD_OPENONLY 0x00002000
#define FMOD_ACCURATETIME 0x00004000
#define FMOD_MPEGSEARCH 0x00008000
#define FMOD_NONBLOCKING 0x00010000
#define FMOD_UNIQUE 0x00020000
#define FMOD_3D_HEADRELATIVE 0x00040000
#define FMOD_3D_WORLDRELATIVE 0x00080000
#define FMOD_3D_INVERSEROLLOFF 0x00100000
#define FMOD_3D_LINEARROLLOFF 0x00200000
#define FMOD_3D_LINEARSQUAREROLLOFF 0x00400000
#define FMOD_3D_CUSTOMROLLOFF 0x04000000
#define FMOD_3D_IGNOREGEOMETRY 0x40000000
#define FMOD_UNICODE 0x10000000
#define FMOD_IGNORETAGS 0x02000000
#define FMOD_LOWMEM 0x08000000
#define FMOD_LOADSECONDARYRAM 0x20000000
#define FMOD_VIRTUAL_PLAYFROMSTART 0x80000000

Values

**FMOD_DEFAULT**

Default for all modes listed below. FMOD_LOOP_OFF, FMOD_2D, FMOD_HARDWARE

**FMOD_LOOP_OFF**

For non looping sounds. (DEFAULT). Overrides FMOD_LOOP_NORMAL /
**FMOD_LOOP_BIDI.**

**FMOD_LOOP_NORMAL**

For forward looping sounds.

**FMOD_LOOP_BIDI**

For bidirectional looping sounds. (only works on software mixed static sounds).

**FMOD_2D**

Ignores any 3d processing. (DEFAULT).

**FMOD_3D**

Makes the sound positionable in 3D. Overrides FMOD_2D.

**FMOD_HARDWARE**

Attempts to make sounds use hardware acceleration. (DEFAULT). Note on platforms that don't support FMOD_HARDWARE (only 3DS, PS Vita, PSP, Wii and Wii U support FMOD_HARDWARE), this will be internally treated as FMOD_SOFTWARE.

**FMOD_SOFTWARE**

Makes the sound be mixed by the FMOD CPU based software mixer. Overrides FMOD_HARDWARE. Use this for FFT, DSP, compressed sample support, 2D multi-speaker support and other software related features.

**FMOD_CREATESTREAM**

Decompress at runtime, streaming from the source provided (ie from disk). Overrides FMOD_CREATESAMPLE and FMOD_CREATECOMPRESSEDSAMPLE. Note a stream can only be played once at a time due to a stream only having 1 stream buffer and file handle. Open multiple streams to have them play concurrently.

**FMOD_CREATESAMPLE**
Decompress at loadtime, decompressing or decoding whole file into memory as the target sample format (ie PCM). Fastest for FMOD_SOFTWARE based playback and most flexible.

**FMOD_CREATECOMPRESSESAMPLE**

Load MP2, MP3, IMAADPCM or XMA into memory and leave it compressed. During playback the FMOD software mixer will decode it in realtime as a 'compressed sample'. Can only be used in combination with FMOD_SOFTWARE. Overrides FMOD_CREATESAMPLE. If the sound data is not ADPCM, MPEG or XMA it will behave as if it was created with FMOD_CREATESAMPLE and decode the sound into PCM.

**FMOD_OPENUSER**

Opens a user created static sample or stream. Use FMOD_CREATESOUNDEXINFO to specify format and/or read callbacks. If a user created 'sample' is created with no read callback, the sample will be empty. Use Sound::lock and Sound::unlock to place sound data into the sound if this is the case.

**FMOD_OPENMEMORY**

"name_or_data" will be interpreted as a pointer to memory instead of filename for creating sounds. Use FMOD_CREATESOUNDEXINFO to specify length. If used with FMOD_CREATESAMPLE or FMOD_CREATECOMPRESSESAMPLE, FMOD duplicates the memory into its own buffers. Your own buffer can be freed after open. If used with FMOD_CREATESTREAM, FMOD will stream out of the buffer whose pointer you passed in. In this case, your own buffer should not be freed until you have finished with and released the stream.

**FMOD_OPENMEMORY_POINT**

"name_or_data" will be interpreted as a pointer to memory instead of filename for creating sounds. Use FMOD_CREATESOUNDEXINFO to specify length. This differs to FMOD_OPENMEMORY in that it uses the memory as is, without duplicating the memory into its own buffers. For Wii/PSP FMOD_HARDWARE supports this flag for the GCADPCM/VAG formats. On other platforms FMOD_SOFTWARE must be used, as sound hardware on the other platforms (ie
PC) cannot access main ram. Cannot be freed after open, only after Sound::release. Will not work if the data is compressed and FMOD_CREATECOMPRESSESAMPLE is not used.

**FMOD_OPENRAW**

Will ignore file format and treat as raw pcm. Use FMOD_CREATESOUNDEXINFO to specify format. Requires at least defaultfrequency, numchannels and format to be specified before it will open. Must be little endian data.

**FMOD_OPENONLY**

Just open the file, dont prebuffer or read. Good for fast opens for info, or when sound::readData is to be used.

**FMOD_ACCURATETIME**

For [System::createSound](#) - for accurate Sound::getLength/Channel::setPosition on VBR MP3, and MOD/S3M/XM/IT/MIDI files. Scans file first, so takes longer to open. FMOD_OPENONLY does not affect this.

**FMOD_MPEGSEARCH**

For corrupted / bad MP3 files. This will search all the way through the file until it hits a valid MPEG header. Normally only searches for 4k.

**FMOD_NONBLOCKING**

For opening sounds and getting streamed subsounds (seeking) asynchronously. Use [Sound::getOpenState](#) to poll the state of the sound as it opens or retrieves the subsound in the background.

**FMOD_UNIQUE**

Unique sound, can only be played one at a time

**FMOD_3D_HEADRELATIVE**

Make the sound's position, velocity and orientation relative to the listener.
**FMOD\_3D\_WORLDRELATIVE**

Make the sound's position, velocity and orientation absolute (relative to the world). (DEFAULT)

**FMOD\_3D\_INVERSEROLLOFF**

This sound will follow the inverse rolloff model where mindistance = full volume, maxdistance = where sound stops attenuating, and rolloff is fixed according to the global rolloff factor. (DEFAULT)

**FMOD\_3D\_LINEARROLLOFF**

This sound will follow a linear rolloff model where mindistance = full volume, maxdistance = silence. Rolloffscale is ignored.

**FMOD\_3D\_LINEARSQUAREROLLOFF**

This sound will follow a linear-square rolloff model where mindistance = full volume, maxdistance = silence. Rolloffscale is ignored.

**FMOD\_3D\_CUSTOMROLLOFF**

This sound will follow a rolloff model defined by `Sound::set3DCustomRolloff` / `Channel::set3DCustomRolloff`.

**FMOD\_3D\_IGNOREGEOMETRY**

Is not affect by geometry occlusion. If not specified in `Sound::setMode`, or `Channel::setMode`, the flag is cleared and it is affected by geometry again.

**FMOD\_UNICODE**

Filename is double-byte unicode.

**FMOD\_IGNORETAGS**

Skips id3v2/asf/etc tag checks when opening a sound, to reduce seek/read overhead when opening files (helps with CD performance).

**FMOD\_LOWMEM**
Removes some features from samples to give a lower memory overhead, like Sound::getName. See remarks.

**FMOD_LOADSECONDARYRAM**

Load sound into the secondary RAM of supported platform. On PS3, sounds will be loaded into RSX/VRAM.

**FMOD_VIRTUAL_PLAYFROMSTART**

For sounds that start virtual (due to being quiet or low importance), instead of swapping back to audible, and playing at the correct offset according to time, this flag makes the sound play from the start.

**See Also**

- System::createSound
- System::createStream
- Sound::setMode
- Sound::getMode
- Channel::setMode
- Channel::getMode
- Sound::set3DCustomRolloff
- Channel::set3DCustomRolloff
- Sound::getOpenState
FMOD_REVERB_CHANNELFLAGS

Values for the Flags member of the FMOD_REVERB_CHANNELPROPERTIES structure. For SFX Reverb, there is support for multiple reverb environments.

Use FMOD_REVERB_CHANNELFLAGS_ENVIRONMENT0 to FMOD_REVERB_CHANNELFLAGS_ENVIRONMENT3 in the flags member of FMOD_REVERB_CHANNELPROPERTIES to specify which environment instance(s) to target.

- If you do not specify any instance the first reverb instance will be used.
- If you specify more than one instance with getReverbProperties, the first instance will be used.
- If you specify more than one instance with setReverbProperties, it will set more than 1 instance at once.

Definition

#define FMOD_REVERB_CHANNELFLAGS_INSTANCE0 0x00000010
#define FMOD_REVERB_CHANNELFLAGS_INSTANCE1 0x00000020
#define FMOD_REVERB_CHANNELFLAGS_INSTANCE2 0x00000040
#define FMOD_REVERB_CHANNELFLAGS_INSTANCE3 0x00000080
#define FMOD_REVERB_CHANNELFLAGS_DEFAULT FMOD_REVERB_CHANNELFLAGS

Values

FMOD_REVERB_CHANNELFLAGS_INSTANCE0

SFX/Wii. Specify channel to target reverb instance 0. Default target.

FMOD_REVERB_CHANNELFLAGS_INSTANCE1

SFX/Wii. Specify channel to target reverb instance 1.

FMOD_REVERB_CHANNELFLAGSINSTANCE2

SFX/Wii. Specify channel to target reverb instance 2.

FMOD_REVERB_CHANNELFLAGSINSTANCE3

SFX/Wii. Specify channel to target reverb instance 3.
SFX. Specify channel to target reverb instance 3.

*FMOD_REVERB_CHANNELFLAGS_DEFAULT*

**See Also**

- [FMOD_REVERB_CHANNELPROPERTIES](#)
FMOD_REVERB_FLAGS

Values for the Flags member of the FMOD_REVERB_PROPERTIES structure.

Definition

```c
#define FMOD_REVERB_FLAGS_HIGHQUALITYREVERB 0x00000400
#define FMOD_REVERB_FLAGS_HIGHQUALITYDPL2REVERB 0x00000800
#define FMOD_REVERB_FLAGS_HARDWAREONLY 0x00001000
#define FMOD_REVERB_FLAGS_DEFAULT 0x00000000
```

Values

**FMOD_REVERB_FLAGS_HIGHQUALITYREVERB**

Wii. Use high quality reverb

**FMOD_REVERB_FLAGS_HIGHQUALITYDPL2REVERB**

Wii. Use high quality DPL2 reverb

**FMOD_REVERB_FLAGS_HARDWAREONLY**

Don't create an SFX reverb for FMOD_SOFTWARE channels, hardware reverb only

**FMOD_REVERB_FLAGS_DEFAULT**

See Also

- FMOD_REVERB_PROPERTIES
Firelight Technologies FMOD Ex
FMOD_REVERB_PRESETS

A set of predefined environment PARAMETERS.
These are used to initialize an FMOD_REVERB_PROPERTIES structure statically.

i.e.
FMOD_REVERB_PROPERTIES prop = FMOD_PRESET_GENERIC;

Definition

#define FMOD_PRESET_OFF { 0, -1, 1.00f, -10000, -10000, 0, 1.
#define FMOD_PRESET_GENERIC { 0, 0, 1.00f, -1000, -100, 0,
#define FMOD_PRESET_PADDDECELL { 0, 1, 1.00f, -1000, -6000,
#define FMOD_PRESET_ROOM { 0, 2, 1.00f, -1000, -454, 0, 0
#define FMOD_PRESET_BATHROOM { 0, 3, 1.00f, -1000, -1200, 0,
#define FMOD_PRESET_STONEROOM { 0, 4, 1.00f, -1000, -6000,
#define FMOD_PRESET_AUDITORIUM { 0, 5, 1.00f, -1000, -300, 0
#define FMOD_PRESET_CONCERTHALL { 0, 6, 1.00f, -1000, -476,
#define FMOD_PRESET_PSP_HALL { 0, 7, 1.00f, -1000, -500,
#define FMOD_PRESET_CAVE { 0, 8, 1.00f, -1000, 0, 0, 2
#define FMOD_PRESET_HANGAR { 0, 9, 1.00f, -1000, -698, 0,
#define FMOD_PRESET_PSP_STUDIO_A { 0, 10, 1.00f, -1000, -1000, 0,
#define FMOD_PRESET_PSP_STUDIO_B { 0, 11, 1.00f, -1000, -4,
#define FMOD_PRESET_PSP_ROOM { 0, 12, 1.00f, -1000, -300, 0,
#define FMOD_PRESET_PSP_STONECORRORIDOR { 0, 13, 1.00f, -1000, -237,
#define FMOD_PRESET_ALLEY { 0, 14, 0.30f, -1000, -270, 0,
#define FMOD_PRESET_FOREST { 0, 15, 0.30f, -1000, -3300, 0,
#define FMOD_PRESET_CITY { 0, 16, 0.50f, -1000, -800, 0, 1
#define FMOD_PRESET_MOUNTAINS { 0, 17, 0.27f, -1000, -2500, 0
#define FMOD_PRESET_QUARRY { 0, 18, 1.00f, -1000, -1000, 0,
#define FMOD_PRESET_PSP_STONE { 0, 19, 0.21f, -1000, -2000, 0,
#define FMOD_PRESET_PSP_PARKINGLOT { 0, 20, 1.00f, -1000, 0,
#define FMOD_PRESET_PSP_SEWERPIPE { 0, 21, 0.80f, -1000, -1000, 0
#define FMOD_PRESET_PSP_UNDERWATER { 0, 22, 1.00f, -1000, -4000,
#define FMOD_PRESET_PSP_PSP_ROOM { 0, 1, 0, 0, 0, 0,
#define FMOD_PRESET_PSP_PSP_STUDIO_A { 0, 2, 0, 0, 0,
#define FMOD_PRESET_PSP_PSP_STUDIO_B { 0, 3, 0, 0, 0,
#define FMOD_PRESET_PSP_PSP_PSP_PSP { 0, 4, 0, 0,
#define FMOD_PRESET_PSP_HALL { 0, 5, 0, 0, 0,
#define FMOD_PRESET_PSP_PSP_PSP { 0, 6, 0, 0,
#define FMOD_PRESET_PSP_PSP_PSP { 0, 7, 0, 0,
#define FMOD_PRESET_PSP_PSP_PSP { 0, 8, 0, 0,
#define FMOD_PRESET_PSP_PSP_PSP { 0, 9, 0, 0,
#define FMOD_PRESET_PSP_PSP_PSP { 0, 10, 0, 0,
#define FMOD_PRESET_PSP_PSP_PSP { 0, 11, 0, 0,
#define FMOD_PRESET_PSP_PSP_PSP { 0, 12, 0, 0,
Values

FMOD_PRESET_OFF
FMOD_PRESET_GENERIC
FMOD_PRESET_PADDEDCELL
FMOD_PRESET_ROOM
FMOD_PRESET_BATHROOM
FMOD_PRESET_LIVINGROOM
FMOD_PRESET_STONEROOM
FMOD_PRESET_AUDITORIUM
FMOD_PRESET_CONCERTHALL
FMOD_PRESET_CAVE
FMOD_PRESET_ARENA
FMOD_PRESET_HANGAR
FMOD_PRESET_CARPETTEDHALLWAY
FMOD_PRESET_HALLWAY
FMOD_PRESET_STONECORRIDOR
FMOD_PRESET_ALLEY
FMOD_PRESET_FOREST
FMOD_PRESET_CITY
FMOD_PRESET_MOUNTAINS
FMOD_PRESET_QUARRY
FMOD_PRESET_PLAIN
FMOD_PRESET_PARKINGLOT
FMOD_PRESET_SEWERPIPE
FMOD_PRESET_UNDERWATER
FMOD_PRESET_PSP_ROOM
FMOD_PRESET_PSP_STUDIO_A
FMOD_PRESET_PSP_STUDIO_B
FMOD_PRESET_PSP_STUDIO_C
FMOD_PRESET_PSP_HALL
FMOD_PRESET_PSP_SPACE
FMOD_PRESET_PSP_ECHO
FMOD_PRESET_PSP_DELAY
FMOD_PRESET_PSPPIPE

See Also

- System::setReverbProperties
Firelight Technologies FMOD Ex
FMOD_TIMEUNIT

List of time types that can be returned by Sound::getLength and used with Channel::setPosition or Channel::getPosition.
FMOD_TIMEUNIT_SENTENCE_MS,
FMOD_TIMEUNIT_SENTENCE_PCM,
FMOD_TIMEUNIT_SENTENCE_PCMBYTES,
FMOD_TIMEUNIT_SENTENCE and
FMOD_TIMEUNIT_SENTENCE_SUBSOUND are only supported by Channel functions. Do not combine flags except FMOD_TIMEUNIT_BUFFERED.

Definition

```c
#define FMOD_TIMEUNIT_MS 0x00000001
#define FMOD_TIMEUNIT_PCM 0x00000002
#define FMOD_TIMEUNIT_PCMBYTES 0x00000004
#define FMOD_TIMEUNIT_RAWBYTES 0x00000008
#define FMOD_TIMEUNIT_PCMFRACTION 0x00000010
#define FMOD_TIMEUNIT_MODORDER 0x00000100
#define FMOD_TIMEUNIT_MODROW 0x00000200
#define FMOD_TIMEUNIT_MODPATTERN 0x00000400
#define FMOD_TIMEUNIT_SENTENCE_MS 0x00010000
#define FMOD_TIMEUNIT_SENTENCE_PCM 0x00020000
#define FMOD_TIMEUNIT_SENTENCE_PCMBYTES 0x00040000
#define FMOD_TIMEUNIT_SENTENCE 0x00080000
#define FMOD_TIMEUNIT_SENTENCE_SUBSOUND 0x00100000
#define FMOD_TIMEUNIT_BUFFERED 0x10000000
```

Values

**FMOD_TIMEUNIT_MS**

Milliseconds.

**FMOD_TIMEUNIT_PCM**

PCM samples, related to milliseconds * samplerate / 1000.

**FMOD_TIMEUNIT_PCMBYTES**
Bytes, related to PCM samples * channels * datawidth (ie 16bit = 2 bytes).

**FMOD_TIMEUNIT_RAWBYTES**

Raw file bytes of (compressed) sound data (does not include headers). Only used by `Sound::getLength` and `Channel::getPosition`.

**FMOD_TIMEUNIT_PCMFRACTION**

Fractions of 1 PCM sample. Unsigned int range 0 to 0xFFFFFFFF. Used for sub-sample granularity for DSP purposes.

**FMOD_TIMEUNIT_MODORDER**

MOD/S3M/XM/IT. Order in a sequenced module format. Use `Sound::getFormat` to determine the PCM format being decoded to.

**FMOD_TIMEUNIT_MODROW**

MOD/S3M/XM/IT. Current row in a sequenced module format. `Sound::getLength` will return the number of rows in the currently playing or seeked to pattern.

**FMOD_TIMEUNIT_MODPATTERN**

MOD/S3M/XM/IT. Current pattern in a sequenced module format. `Sound::getLength` will return the number of patterns in the song and `Channel::getPosition` will return the currently playing pattern.

**FMOD_TIMEUNIT_SENTENCE_MS**

Currently playing subsound in a sentence time in milliseconds.

**FMOD_TIMEUNIT_SENTENCE_PCM**

Currently playing subsound in a sentence time in PCM Samples, related to milliseconds * samplerate / 1000.

**FMOD_TIMEUNIT_SENTENCE_PCMBYTES**

Currently playing subsound in a sentence time in bytes, related to PCM samples
* channels * datawidth (ie 16bit = 2 bytes).

**FMOD_TIMEUNIT_SENTENCE**

Currently playing sentence index according to the channel.

**FMOD_TIMEUNIT_SENTENCE_SUBSOUND**

Currently playing subsound index in a sentence.

**FMOD_TIMEUNIT_BUFFERED**

Time value as seen by buffered stream. This is always ahead of audible time, and is only used for processing.

**See Also**

- [Sound::getLength](#)
- [Channel::setPosition](#)
- [Channel::getPosition](#)
Firelight Technologies FMOD Ex
Enumerations

FMOD_CHANNELINDEX FMOD_CHANNEL_CALLBACKTYPE
FMOD_DELAYTYPE
FMOD_DSP_CHORUS
FMOD_DSP_COMPRESSOR
FMOD_DSP_DELAY
FMOD_DSP_DISTORTION
FMOD_DSP_ECHO
FMOD_DSP_FFT_WINDOW
FMOD_DSP_FLANGE
FMOD_DSP_HIGHPASS
FMOD_DSP_HIGHPASS_SIMPLE
FMOD_DSP_ITECHO
FMOD_DSP_ITLOWPASS
FMOD_DSP_LOWPASS
FMOD_DSP_LOWPASS_SIMPLE
FMOD_DSP_NORMALIZE
FMOD_DSP_OSCILLATOR
FMOD_DSP_PARAMEQ
FMOD_DSP_PITCHSHIFT
FMOD_DSP_RESAMPLER
FMOD_DSP_SFXREVERB
FMOD_DSP_TREMOLO
FMOD_DSP_TYPE
FMOD_EVENTPROPERTY_TYPE
FMOD_OPENSTATE
FMOD_OUTPUTTYPE
FMOD_PLUGINTYPE
FMOD_PSP_OUTPUTMODE
FMOD_RESULT
FMOD_SOUNDDGROUP_BEHAVIOR
FMOD_SOUND_FORMAT
FMOD_SOUND_TYPE
FMOD_SPEAKER
FMOD_SPEAKERMAPTYPE
Firelight Technologies FMOD Ex
**FMOD_CHANNELINDEX**

Special channel index values for FMOD functions.

**Enumeration**

```c
typedef enum {
    FMOD_CHANNEL_FREE,
    FMOD_CHANNEL_REUSE
} FMOD_CHANNELINDEX;
```

**Values**

*FMOD_CHANNEL_FREE*

For a channel index, FMOD chooses a free voice using the priority system.

*FMOD_CHANNEL_REUSE*

For a channel index, re-use the channel handle that was passed in.

**Remarks**

To get 'all' of the channels, use `System::getMasterChannelGroup`.

**See Also**

- `System::playSound`
- `System::playDSP`
- `System::getChannel`
- `System::getMasterChannelGroup`
Firelight Technologies FMOD Ex
These callback types are used with `Channel::setCallback`.

### Enumeration

```c
typedef enum {
    FMOD_CHANNEL_CALLBACKTYPE_END,
    FMOD_CHANNEL_CALLBACKTYPE_VIRTUALVOICE,
    FMOD_CHANNEL_CALLBACKTYPE_SYNCPOINT,
    FMOD_CHANNEL_CALLBACKTYPE_OCCLUSION,
    FMOD_CHANNEL_CALLBACKTYPE_MAX
} FMOD_CHANNEL_CALLBACKTYPE;
```

### Values

**FMOD_CHANNEL_CALLBACKTYPE_END**

Called when a sound ends.

**FMOD_CHANNEL_CALLBACKTYPE_VIRTUALVOICE**

Called when a voice is swapped out or swapped in.

**FMOD_CHANNEL_CALLBACKTYPE_SYNCPOINT**

Called when a syncpoint is encountered. Can be from wav file markers.

**FMOD_CHANNEL_CALLBACKTYPE_OCCLUSION**

Called when the channel has its geometry occlusion value calculated. Can be used to clamp or change the value.

**FMOD_CHANNEL_CALLBACKTYPE_MAX**

Maximum number of callback types supported.

### Remarks
Each callback has commanddata parameters passed as int unique to the type of callback.
See reference to **FMOD_CHANNEL_CALLBACK** to determine what they might mean for each type of callback.

**Note!** Currently the user must call **System::update** for these callbacks to trigger!

**See Also**

- [Channel::setCallback](#)
- [FMOD_CHANNEL_CALLBACK](#)
- [System::update](#)
Firelight Technologies FMOD Ex
FMOD_DELAYTYPE

Types of delay that can be used with Channel::setDelay / Channel::getDelay.

Enumeration

typedef enum {
    FMOD_DELAYTYPE_END_MS,
    FMOD_DELAYTYPE_DSPCLOCK_START,
    FMOD_DELAYTYPE_DSPCLOCK_END,
    FMOD_DELAYTYPE_DSPCLOCK_PAUSE,
    FMOD_DELAYTYPE_MAX
} FMOD_DELAYTYPE;

Values

FMOD_DELAYTYPE_END_MS

Delay at the end of the sound in milliseconds. Use delayhi only. Channel::isPlaying will remain true until this delay has passed even though the sound itself has stopped playing.

FMOD_DELAYTYPE_DSPCLOCK_START

Time the sound started if Channel::getDelay is used, or if Channel::setDelay is used, the sound will delay playing until this exact tick.

FMOD_DELAYTYPE_DSPCLOCK_END

Time the sound should end. If this is non-zero, the channel will go silent at this exact tick.

FMOD_DELAYTYPE_DSPCLOCK_PAUSE

Time the sound should pause. If this is non-zero, the channel will pause at this exact tick.

FMOD_DELAYTYPE_MAX
Maximum number of tag datatypes supported.

**Remarks**

If you haven't called `Channel::setDelay` yet, if you call `Channel::getDelay` with `FMOD_DELAYTYPE_DSPCLOCK_START` it will return the equivalent global DSP clock value to determine when a channel started, so that you can use it for other channels to sync against.

Use `System::getDSPClock` to also get the current dspclock time, a base for future calls to `Channel::setDelay`.

Use `FMOD_64BIT_ADD` or `FMOD_64BIT_SUB` to add a hi/lo combination together and cope with wraparound.
If `FMOD_DELAYTYPE_END_MS` is specified, the value is not treated as a 64 bit number, just the delayhi value is used and it is treated as milliseconds.

**See Also**

- `Channel::setDelay`
- `Channel::getDelay`
- `System::getDSPClock`
Firelight Technologies FMOD Ex
FMOD_DSP_CHORUS

Parameter types for the FMOD_DSP_TYPE_CHORUS filter.

Enumeration

typedef enum {
   FMOD_DSP_CHORUS_DRYMIX,
   FMOD_DSP_CHORUS_WETMIX1,
   FMOD_DSP_CHORUS_WETMIX2,
   FMOD_DSP_CHORUS_WETMIX3,
   FMOD_DSP_CHORUS_DELAY,
   FMOD_DSP_CHORUS_RATE,
   FMOD_DSP_CHORUS_DEPTH
} FMOD_DSP_CHORUS;

Values

FMOD_DSP_CHORUS_DRYMIX

Volume of original signal to pass to output. 0.0 to 1.0. Default = 0.5.

FMOD_DSP_CHORUS_WETMIX1

Volume of 1st chorus tap. 0.0 to 1.0. Default = 0.5.

FMOD_DSP_CHORUS_WETMIX2

Volume of 2nd chorus tap. This tap is 90 degrees out of phase of the first tap. 0.0 to 1.0. Default = 0.5.

FMOD_DSP_CHORUS_WETMIX3

Volume of 3rd chorus tap. This tap is 90 degrees out of phase of the second tap. 0.0 to 1.0. Default = 0.5.

FMOD_DSP_CHORUS_DELAY

Chorus delay in ms. 0.1 to 100.0. Default = 40.0 ms.
**FMOD_DSP_CHORUS_RATE**

Chorus modulation rate in hz. 0.0 to 20.0. Default = 0.8 hz.

**FMOD_DSP_CHORUS_DEPTH**

Chorus modulation depth. 0.0 to 1.0. Default = 0.03.

**Remarks**

Chorus is an effect where the sound is more 'spacious' due to 1 to 3 versions of the sound being played along side the original signal but with the pitch of each copy modulating on a sine wave.

**See Also**

- [DSP::setParameter](#)
- [DSP::getParameter](#)
- [FMOD_DSP_TYPE](#)
Firelight Technologies FMOD Ex
**FMOD_DSP_COMPRESSOR**

Parameter types for the `FMOD_DSP_TYPE_COMPRESSOR` unit. This is a simple linked multichannel software limiter that is uniform across the whole spectrum.

**Enumeration**

```c
typedef enum {
    FMOD_DSP_COMPRESSOR_THRESHOLD,
    FMOD_DSP_COMPRESSOR_ATTACK,
    FMOD_DSP_COMPRESSOR_RELEASE,
    FMOD_DSP_COMPRESSOR_GAINMAKEUP
} FMOD_DSP_COMPRESSOR;
```

**Values**

**FMOD_DSP_COMPRESSOR_THRESHOLD**

Threshold level (dB) in the range from -60 through 0. The default value is 0.

**FMOD_DSP_COMPRESSOR_ATTACK**

Gain reduction attack time (milliseconds), in the range from 10 through 200. The default value is 50.

**FMOD_DSP_COMPRESSOR_RELEASE**

Gain reduction release time (milliseconds), in the range from 20 through 1000. The default value is 50.

**FMOD_DSP_COMPRESSOR_GAINMAKEUP**

Make-up gain (dB) applied after limiting, in the range from 0 through 30. The default value is 0.
Remarks

The limiter is not guaranteed to catch every peak above the threshold level, because it cannot apply gain reduction instantaneously - the time delay is determined by the attack time. However setting the attack time too short will distort the sound, so it is a compromise. High level peaks can be avoided by using a short attack time - but not too short, and setting the threshold a few decibels below the critical level.

See Also

- DSP::SetParameter
- DSP::GetParameter
- FMOD_DSP_TYPE
- System::addDSP
Firelight Technologies FMOD Ex
**FMOD_DSP_DELAY**

Parameter types for the **FMOD_DSP_TYPE_DELAY** filter.

**Enumeration**

```c
typedef enum {
    FMOD_DSP_DELAY_CH0,
    FMOD_DSP_DELAY_CH1,
    FMOD_DSP_DELAY_CH2,
    FMOD_DSP_DELAY_CH3,
    FMOD_DSP_DELAY_CH4,
    FMOD_DSP_DELAY_CH5,
    FMOD_DSP_DELAY_CH6,
    FMOD_DSP_DELAY_CH7,
    FMOD_DSP_DELAY_CH8,
    FMOD_DSP_DELAY_CH9,
    FMOD_DSP_DELAY_CH10,
    FMOD_DSP_DELAY_CH11,
    FMOD_DSP_DELAY_CH12,
    FMOD_DSP_DELAY_CH13,
    FMOD_DSP_DELAY_CH14,
    FMOD_DSP_DELAY_CH15,
    FMOD_DSP_DELAY_MAX_DELAY
} FMOD_DSP_DELAY;
```

**Values**

**FMOD_DSP_DELAY_CH0**

Channel #0 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH1**

Channel #1 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH2**

Channel #2 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH3**
Channel #3 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH4*

Channel #4 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH5*

Channel #5 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH6*

Channel #6 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH7*

Channel #7 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH8*

Channel #8 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH9*

Channel #9 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH10*

Channel #10 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH11*

Channel #11 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH12*

Channel #12 Delay in ms. 0 to 10000. Default = 0.

*FMOD_DSP_DELAY_CH13*

Channel #13 Delay in ms. 0 to 10000. Default = 0.
**FMOD_DSP_DELAY_CH14**

Channel #14 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH15**

Channel #15 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_MAXDELAY**

Maximum delay in ms. 0 to 10000. Default = 10.

**Remarks**

Note. Every time MaxDelay is changed, the plugin re-allocates the delay buffer. This means the delay will disappear at that time while it refills its new buffer. A larger MaxDelay results in larger amounts of memory allocated. Channel delays above MaxDelay will be clipped to MaxDelay and the delay buffer will not be resized.

**See Also**

- [DSP::setParameter](#)
- [DSP::getParameter](#)
- [FMOD_DSP_TYPE](#)
Firelight Technologies FMOD Ex
FMOD_DSP_DISTORTION

Parameter types for the FMOD_DSP_TYPE_DISTORTION filter.

Enumeration

typedef enum {
   FMOD_DSP_DISTORTION_LEVEL
} FMOD_DSP_DISTORTION;

Values

FMOD_DSP_DISTORTION_LEVEL

Distortion value. 0.0 to 1.0. Default = 0.5.

See Also

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
**FMOD_DSP_ECHO**

Parameter types for the [FMOD_DSP_TYPE_ECHO](#) filter.

**Enumeration**

```c
typedef enum {
    FMOD_DSP_ECHO_DELAY,
    FMOD_DSP_ECHO_DECAYRATIO,
    FMOD_DSP_ECHO_MAXCHANNELS,
    FMOD_DSP_ECHO_DRYMIX,
    FMOD_DSP_ECHO_WETMIX
} FMOD_DSP_ECHO;
```

**Values**

**FMOD_DSP_ECHO_DELAY**

Echo delay in ms. 10 to 5000. Default = 500.

**FMOD_DSP_ECHO_DECAYRATIO**

Echo decay per delay. 0 to 1. 1.0 = No decay, 0.0 = total decay (ie simple 1 line delay). Default = 0.5.

**FMOD_DSP_ECHO_MAXCHANNELS**

Maximum channels supported. 0 to 16. 0 = same as fmod's default output polyphony, 1 = mono, 2 = stereo etc. See remarks for more. Default = 0. It is suggested to leave at 0!

**FMOD_DSP_ECHO_DRYMIX**

Volume of original signal to pass to output. 0.0 to 1.0. Default = 1.0.

**FMOD_DSP_ECHO_WETMIX**

Volume of echo signal to pass to output. 0.0 to 1.0. Default = 1.0.
Remarks

Note. Every time the delay is changed, the plugin re-allocates the echo buffer. This means the echo will dissapear at that time while it refills its new buffer. Larger echo delays result in larger amounts of memory allocated.

'maxchannels' also dictates the amount of memory allocated. By default, the maxchannels value is 0. If FMOD is set to stereo, the echo unit will allocate enough memory for 2 channels. If it is 5.1, it will allocate enough memory for a 6 channel echo, etc.
If the echo effect is only ever applied to the global mix (ie it was added with System::addDSP), then 0 is the value to set as it will be enough to handle all speaker modes.
When the echo is added to a channel (ie Channel::addDSP) then the channel count that comes in could be anything from 1 to 8 possibly. It is only in this case where you might want to increase the channel count above the output's channel count.
If a channel echo is set to a lower number than the sound's channel count that is coming in, it will not echo the sound.

See Also

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
**FMOD_DSP_FFT_WINDOW**

List of windowing methods used in spectrum analysis to reduce leakage / transient signals interfering with the analysis.

This is a problem with analysis of continuous signals that only have a small portion of the signal sample (the fft window size).

Windowing the signal with a curve or triangle tapers the sides of the fft window to help alleviate this problem.

**Enumeration**

```c
typedef enum {
    FMOD_DSP_FFT_WINDOW_RECT,
    FMOD_DSP_FFT_WINDOW_TRIANGLE,
    FMOD_DSP_FFT_WINDOW_HAMMING,
    FMOD_DSP_FFT_WINDOW_HANNING,
    FMOD_DSP_FFT_WINDOW_BLACKMAN,
    FMOD_DSP_FFT_WINDOW_BLACKMANHARRIS,
    FMOD_DSP_FFT_WINDOW_MAX
} FMOD_DSP_FFT_WINDOW;
```

**Values**

**FMOD_DSP_FFT_WINDOW_RECT**

w[n] = 1.0

**FMOD_DSP_FFT_WINDOW_TRIANGLE**

w[n] = TRI(2n/N)

**FMOD_DSP_FFT_WINDOW_HAMMING**

w[n] = 0.54 - (0.46 * COS(n/N) )

**FMOD_DSP_FFT_WINDOW_HANNING**

w[n] = 0.5 * (1.0 - COS(n/N) )

**FMOD_DSP_FFT_WINDOW_BLACKMAN**
\[ w[n] = 0.42 - (0.5 \times \cos(n/N)) + (0.08 \times \cos(2.0 \times n/N)) \]

**FMOD_DSP_FFT_WINDOW_BLACKMANHARRIS**

\[ w[n] = 0.35875 - (0.48829 \times \cos(1.0 \times n/N)) + (0.14128 \times \cos(2.0 \times n/N)) - (0.01168 \times \cos(3.0 \times n/N)) \]

**FMOD_DSP_FFT_WINDOW_MAX**

Maximum number of FFT window types supported.

**Remarks**

Cyclic signals such as a sine wave that repeat their cycle in a multiple of the window size do not need windowing. I.e. If the sine wave repeats every 1024, 512, 256 etc samples and the FMOD fft window is 1024, then the signal would not need windowing. Not windowing is the same as **FMOD_DSP_FFT_WINDOW_RECT**, which is the default. If the cycle of the signal (ie the sine wave) is not a multiple of the window size, it will cause frequency abnormalities, so a different windowing method is needed.

**FMOD_DSP_FFT_WINDOW_RECT.**

![Image of FMOD_DSP_FFT_WINDOW_RECT](image)

**FMOD_DSP_FFT_WINDOW_TRIANGLE.**

![Image of FMOD_DSP_FFT_WINDOW_TRIANGLE](image)

**FMOD_DSP_FFT_WINDOW_HAMMING.**

![Image of FMOD_DSP_FFT_WINDOW_HAMMING](image)
See Also

- System::getSpectrum
- Channel::getSpectrum
Firelight Technologies FMOD Ex
FMOD_DSP_FLANGE

Parameter types for the FMOD_DSP_TYPE_FLANGE filter.

Enumeration

typedef enum {
  FMOD_DSP_FLANGE_DRYMIX,
  FMOD_DSP_FLANGE_WETMIX,
  FMOD_DSP_FLANGE_DEPTH,
  FMOD_DSP_FLANGE_RATE
} FMOD_DSP_FLANGE;

Values

FMOD_DSP_FLANGE_DRYMIX

Volume of original signal to pass to output. 0.0 to 1.0. Default = 0.45.

FMOD_DSP_FLANGE_WETMIX

Volume of flange signal to pass to output. 0.0 to 1.0. Default = 0.55.

FMOD_DSP_FLANGE_DEPTH

Flange depth (percentage of 40ms delay). 0.01 to 1.0. Default = 1.0.

FMOD_DSP_FLANGE_RATE

Flange speed in hz. 0.0 to 20.0. Default = 0.1.

Remarks

Flange is an effect where the signal is played twice at the same time, and one copy slides back and forth creating a whooshing or flanging effect. As there are 2 copies of the same signal, by default each signal is given 50% mix, so that the total is not louder than the original unaffected signal.
Flange depth is a percentage of a 10ms shift from the original signal. Anything above 10ms is not considered flange because to the ear it begins to 'echo' so 10ms is the highest value possible.

See Also

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
FMOD_DSP_HIGHPASS

Parameter types for the FMOD_DSP_TYPE_HIGHPASS filter.

Enumeration

typedef enum {
    FMOD_DSP_HIGHPASS_CUTOFF,
    FMOD_DSP_HIGHPASS_RESONANCE
} FMOD_DSP_HIGHPASS;

Values

FMOD_DSP_HIGHPASS_CUTOFF

Highpass cutoff frequency in hz. 1.0 to output 22000.0. Default = 5000.0.

FMOD_DSP_HIGHPASS_RESONANCE

Highpass resonance Q value. 1.0 to 10.0. Default = 1.0.

See Also

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
FMOD_DSP_HIGHPASS_SIMPLE

Parameter types for the \texttt{FMOD_DSP_TYPE\_HIGHPASS\_SIMPLE} filter. This is a very simple single-order high pass filter. The emphasis is on speed rather than accuracy, so this should not be used for task requiring critical filtering.

\textbf{Enumeration}

\begin{verbatim}
typedef enum {
    FMOD_DSP_HIGHPASS_SIMPLE_CUTOFF
} FMOD_DSP_HIGHPASS_SIMPLE;
\end{verbatim}

\textbf{Values}

\texttt{FMOD_DSP_HIGHPASS_SIMPLE\_CUTOFF}

Highpass cutoff frequency in hz. 10.0 to 22000.0. Default = 1000.0

\textbf{See Also}

- \texttt{DSP::setParameter}
- \texttt{DSP::getParameter}
- \texttt{FMOD_DSP_TYPE}
Firelight Technologies FMOD Ex
FMOD_DSP_ITECHO

Parameter types for the FMOD_DSP_TYPE_ITECHO filter.
This is effectively a software based echo filter that emulates the DirectX DMO echo effect. Impulse tracker files can support this, and FMOD will produce the effect on ANY platform, not just those that support DirectX effects!

Enumeration

typedef enum {
    FMOD_DSP_ITECHO_WETDRY MIX,
    FMOD_DSP_ITECHO_FEED BACK,
    FMOD_DSP_ITECHO_LEFTDE LAY,
    FMOD_DSP_ITECHO_RIGHT DE LAY,
    FMOD_DSP_ITECHO_PANDE LAY
} FMOD_DSP_ITECHO;

Values

FMOD_DSP_ITECHO_WETDRY MIX

Ratio of wet (processed) signal to dry (unprocessed) signal. Must be in the range from 0.0 through 100.0 (all wet). The default value is 50.

FMOD_DSP_ITECHO_FEED BACK

Percentage of output fed back into input, in the range from 0.0 through 100.0. The default value is 50.

FMOD_DSP_ITECHO_LEFTDELAY

Delay for left channel, in milliseconds, in the range from 1.0 through 2000.0. The default value is 500 ms.

FMOD_DSP_ITECHO_RIGHTDELAY

Delay for right channel, in milliseconds, in the range from 1.0 through 2000.0. The default value is 500 ms.
**FMOD_DSP_ITECHO_PANDELAY**

Value that specifies whether to swap left and right delays with each successive echo. The default value is zero, meaning no swap. Possible values are defined as 0.0 (equivalent to FALSE) and 1.0 (equivalent to TRUE). CURRENTLY NOT SUPPORTED.

**Remarks**

Note. Every time the delay is changed, the plugin re-allocates the echo buffer. This means the echo will dissapear at that time while it refills its new buffer. Larger echo delays result in larger amounts of memory allocated.

As this is a stereo filter made mainly for IT playback, it is targeted for stereo signals. With mono signals only the **FMOD_DSP_ITECHO_LEFTDELAY** is used. For multichannel signals (>2) there will be no echo on those channels.

**See Also**

- DSP::SetParameter
- DSP::GetParameter
- FMOD_DSP_TYPE
- System::addDSP
Firelight Technologies FMOD Ex
FMOD_DSP_ITLOWPASS

Parameter types for the FMOD_DSP_TYPE_ITLOWPASS filter.
This is different to the default FMOD_DSP_TYPE_ITLOWPASS filter in that
it uses a different quality algorithm and is the filter used to produce the correct
sounding playback in .IT files.
FMOD Ex's .IT playback uses this filter.

Enumeration

typedef enum {
    FMOD_DSP_ITLOWPASS_CUTOFF,
    FMOD_DSP_ITLOWPASS_RESONANCE
} FMOD_DSP_ITLOWPASS;

Values

FMOD_DSP_ITLOWPASS_CUTOFF
Lowpass cutoff frequency in hz. 1.0 to 22000.0. Default = 5000.0/

FMOD_DSP_ITLOWPASS_RESONANCE
Lowpass resonance Q value. 0.0 to 127.0. Default = 1.0.

Remarks

Note! This filter actually has a limited cutoff frequency below the specified
maximum, due to its limited design, so for a more open range filter use
FMOD_DSP_LOWPASS or if you don't mind not having resonance,
FMOD_DSP_LOWPASS_SIMPLE.
The effective maximum cutoff is about 8060hz.

See Also
• DSP::setParameter
• DSP::getParameter
• FMODE_DSP_TYPE
Firelight Technologies FMOD Ex
FMOD_DSP_LOWPASS

Parameter types for the FMOD_DSP_TYPE_LOWPASS filter.

Enumeration

typedef enum {
   FMOD_DSP_LOWPASS_CUTOFF,
   FMOD_DSP_LOWPASS_RESONANCE
} FMOD_DSP_LOWPASS;

Values

FMOD_DSP_LOWPASS_CUTOFF

Lowpass cutoff frequency in hz. 10.0 to 22000.0. Default = 5000.0.

FMOD_DSP_LOWPASS_RESONANCE

Lowpass resonance Q value. 1.0 to 10.0. Default = 1.0.

See Also

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
FMOD_DSP_LOWPASS_SIMPLE

Parameter types for the `FMOD_DSP_TYPE_LOWPASS_SIMPLE` filter. This is a very simple low pass filter, based on two single-pole RC time-constant modules. The emphasis is on speed rather than accuracy, so this should not be used for task requiring critical filtering.

Enumeration

```c
typedef enum {
    FMOD_DSP_LOWPASS_SIMPLE_CUTOFF
} FMOD_DSP_LOWPASS_SIMPLE;
```

Values

`FMOD_DSP_LOWPASS_SIMPLE_CUTOFF`

Lowpass cutoff frequency in hz. 10.0 to 22000.0. Default = 5000.0

See Also

- `DSP::setParameter`
- `DSP::getParameter`
- `FMOD_DSP_TYPE`
Firelight Technologies FMOD Ex
**FMOD_DSP_NORMALIZE**

Parameter types for the **FMOD_DSP_TYPE_NORMALIZE** filter.

**Enumeration**

typedef enum {
  FMOD_DSP_NORMALIZE_FADETIME,
  FMOD_DSP_NORMALIZE_THRESHHOLD,
  FMOD_DSP_NORMALIZE_MAXAMP
} FMOD_DSP_NORMALIZE;

**Values**

**FMOD_DSP_NORMALIZE_FADETIME**

Time to ramp the silence to full in ms. 0.0 to 20000.0. Default = 5000.0.

**FMOD_DSP_NORMALIZE_THRESHHOLD**

Lower volume range threshold to ignore. 0.0 to 1.0. Default = 0.1. Raise higher to stop amplification of very quiet signals.

**FMOD_DSP_NORMALIZE_MAXAMP**

Maximum amplification allowed. 1.0 to 100000.0. Default = 20.0. 1.0 = no amplification, higher values allow more boost.

**Remarks**

Normalize amplifies the sound based on the maximum peaks within the signal. For example if the maximum peaks in the signal were 50% of the bandwidth, it would scale the whole sound by 2.

The lower threshold value makes the normalizer ignores peaks below a certain point, to avoid over-amplification if a loud signal suddenly came in, and also to avoid amplifying to maximum things like background hiss.
Because FMOD is a realtime audio processor, it doesn't have the luxury of knowing the peak for the whole sound (ie it can't see into the future), so it has to process data as it comes in.
To avoid very sudden changes in volume level based on small samples of new data, fmod fades towards the desired amplification which makes for smooth gain control. The fadetime parameter can control this.

See Also

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
FMOD_DSP_OSCILLATOR

Parameter types for the FMOD_DSP_TYPE_OSCILLATOR filter.

Enumeration

typedef enum {
  FMOD_DSP_OSCILLATOR_TYPE,
  FMOD_DSP_OSCILLATOR_RATE
} FMOD_DSP_OSCILLATOR;

Values

FMOD_DSP_OSCILLATOR_TYPE

Waveform type. 0 = sine. 1 = square. 2 = sawup. 3 = sawdown. 4 = triangle. 5 = noise.

FMOD_DSP_OSCILLATOR_RATE

Frequency of the sinewave in hz. 1.0 to 22000.0. Default = 220.0.

See Also

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
FMOD_DSP_PARAMEQ

Parameter types for the FMOD_DSP_TYPE_PARAMEQ filter.

Enumeration

typedef enum {
    FMOD_DSP_PARAMEQ_CENTER,
    FMOD_DSP_PARAMEQ_BANDWIDTH,
    FMOD_DSP_PARAMEQ_GAIN
} FMOD_DSP_PARAMEQ;

Values

FMOD_DSP_PARAMEQ_CENTER
Frequency center. 20.0 to 22000.0. Default = 8000.0.

FMOD_DSP_PARAMEQ_BANDWIDTH
Octave range around the center frequency to filter. 0.2 to 5.0. Default = 1.0.

FMOD_DSP_PARAMEQ_GAIN
Frequency Gain. 0.05 to 3.0. Default = 1.0.

Remarks

Parametric EQ is a bandpass filter that attenuates or amplifies a selected frequency and its neighbouring frequencies.

To create a multi-band EQ create multiple FMOD_DSP_TYPE_PARAMEQ units and set each unit to different frequencies, for example 1000hz, 2000hz, 4000hz, 8000hz, 16000hz with a range of 1 octave each.

When a frequency has its gain set to 1.0, the sound will be unaffected and represents the original signal exactly.
See Also

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
FMOD_DSP_PITCHSHIFT

Parameter types for the FMOD_DSP_TYPE_PITCHSHIFT filter.

Enumeration

typedef enum {
    FMOD_DSP_PITCHSHIFT_PITCH,
    FMOD_DSP_PITCHSHIFT_FFTSIZE,
    FMOD_DSP_PITCHSHIFT_OVERLAP,
    FMOD_DSP_PITCHSHIFT_MAXCHANNELS
} FMOD_DSP_PITCHSHIFT;

Values

FMOD_DSP_PITCHSHIFT_PITCH

Pitch value. 0.5 to 2.0. Default = 1.0. 0.5 = one octave down, 2.0 = one octave up. 1.0 does not change the pitch.

FMOD_DSP_PITCHSHIFT_FFTSIZE

FFT window size. 256, 512, 1024, 2048, 4096. Default = 1024. Increase this to reduce 'smearing'. This effect is a warbling sound similar to when an mp3 is encoded at very low bitrates.

FMOD_DSP_PITCHSHIFT_OVERLAP

Removed. Do not use. FMOD now uses 4 overlaps and cannot be changed.

FMOD_DSP_PITCHSHIFT_MAXCHANNELS

Maximum channels supported. 0 to 16. 0 = same as fmod's default output polyphony, 1 = mono, 2 = stereo etc. See remarks for more. Default = 0. It is suggested to leave at 0!

Remarks
This pitch shifting unit can be used to change the pitch of a sound without speeding it up or slowing it down. It can also be used for time stretching or scaling, for example if the pitch was doubled, and the frequency of the sound was halved, the pitch of the sound would sound correct but it would be twice as slow.

**Warning!** This filter is very computationally expensive! Similar to a vocoder, it requires several overlapping FFT and IFFT's to produce smooth output, and can require around 440mhz for 1 stereo 48khz signal using the default settings. Reducing the signal to mono will half the cpu usage. Reducing this will lower audio quality, but what settings to use are largely dependant on the sound being played. A noisy polyphonic signal will need higher fft size compared to a speaking voice for example.

This pitch shifter is based on the pitch shifter code at http://www.dspdimension.com, written by Stephan M. Bernsee. The original code is COPYRIGHT 1999-2003 Stephan M. Bernsee.

'maxchannels' dictates the amount of memory allocated. By default, the maxchannels value is 0. If FMOD is set to stereo, the pitch shift unit will allocate enough memory for 2 channels. If it is 5.1, it will allocate enough memory for a 6 channel pitch shift, etc. If the pitch shift effect is only ever applied to the global mix (ie it was added with System::addDSP), then 0 is the value to set as it will be enough to handle all speaker modes.
When the pitch shift is added to a channel (ie Channel::addDSP) then the channel count that comes in could be anything from 1 to 8 possibly. It is only in this case where you might want to increase the channel count above the output's channel count.
If a channel pitch shift is set to a lower number than the sound's channel count that is coming in, it will not pitch shift the sound.

**See Also**

- DSP::setParameter
- DSP::getParameter
- FMOD_DSP_TYPE
Firelight Technologies FMOD Ex
**FMOD_DSP_RESAMPLER**

List of interpolation types that the FMOD Ex software mixer supports.

**Enumeration**

```c
typedef enum {
    FMOD_DSP_RESAMPLER_NOINTERP,
    FMOD_DSP_RESAMPLER_LINEAR,
    FMOD_DSP_RESAMPLER_CUBIC,
    FMOD_DSP_RESAMPLER_SPLINE,
    FMOD_DSP_RESAMPLER_MAX
} FMOD_DSP_RESAMPLER;
```

**Values**

*FMOD_DSP_RESAMPLER_NOINTERP*

No interpolation. High frequency aliasing hiss will be audible depending on the sample rate of the sound.

*FMOD_DSP_RESAMPLER_LINEAR*

Linear interpolation (default method). Fast and good quality, causes very slight lowpass effect on low frequency sounds.

*FMOD_DSP_RESAMPLER_CUBIC*

Cubic interpolation. Slower than linear interpolation but better quality.

*FMOD_DSP_RESAMPLER_SPLINE*

5 point spline interpolation. Slowest resampling method but best quality.

*FMOD_DSP_RESAMPLER_MAX*

Maximum number of resample methods supported.
Remarks

The default resampler type is `FMOD_DSP_RESAMPLER_LINEAR`. Use `System::setSoftwareFormat` to tell FMOD the resampling quality you require for `FMOD_SOFTWARE` based sounds.

See Also

- `System::setSoftwareFormat`
- `System::getSoftwareFormat`
Firelight Technologies FMOD Ex
**FMOD_DSP_SFXREVERB**

Parameter types for the **FMOD_DSP_TYPE_SFXREVERB** unit.

**Enumeration**

```c
typedef enum {
    FMOD_DSP_SFXREVERB_DRYLEVEL,
    FMOD_DSP_SFXREVERB_ROOM,
    FMOD_DSP_SFXREVERB_ROOMHF,
    FMOD_DSP_SFXREVERB_DECAYTIME,
    FMOD_DSP_SFXREVERB_DECAYHFRATIO,
    FMOD_DSP_SFXREVERB_REFLECTIONSLEVEL,
    FMOD_DSP_SFXREVERB_REFLECTIONSDELAY,
    FMOD_DSP_SFXREVERB_REVERBLEVEL,
    FMOD_DSP_SFXREVERB_REVERBDELAY,
    FMOD_DSP_SFXREVERB_DIFFUSION,
    FMOD_DSP_SFXREVERB_DENSITY,
    FMOD_DSP_SFXREVERB_HFREFERENCE,
    FMOD_DSP_SFXREVERB_ROOMLF,
    FMOD_DSP_SFXREVERB_LFREFERENCE
} FMOD_DSP_SFXREVERB;
```

**Values**

**`FMOD_DSP_SFXREVERB_DRYLEVEL`**

Dry Level : Mix level of dry signal in output in mB. Ranges from -10000.0 to 0.0. Default is 0.

**`FMOD_DSP_SFXREVERB_ROOM`**

Room : Room effect level at low frequencies in mB. Ranges from -10000.0 to 0.0. Default is -10000.0.

**`FMOD_DSP_SFXREVERB_ROOMHF`**

Room HF : Room effect high-frequency level re. low frequency level in mB. Ranges from -10000.0 to 0.0. Default is 0.0.
**FMOD_DSP_SFXREVERB_DECAYTIME**

Decay Time: Reverberation decay time at low-frequencies in seconds. Ranges from 0.1 to 20.0. Default is 1.0.

**FMOD_DSP_SFXREVERB_DECAYHFRATIO**

Decay HF Ratio: High-frequency to low-frequency decay time ratio. Ranges from 0.1 to 2.0. Default is 0.5.

**FMOD_DSP_SFXREVERB_REFLECTIONSLEVEL**

Reflections: Early reflections level relative to room effect in mB. Ranges from -10000.0 to 1000.0. Default is -10000.0.

**FMOD_DSP_SFXREVERB_REFLECTIONSDELAY**

Reflect Delay: Delay time of first reflection in seconds. Ranges from 0.0 to 0.3. Default is 0.02.

**FMOD_DSP_SFXREVERB_REVERBLEVEL**

Reverb: Late reverberation level relative to room effect in mB. Ranges from -10000.0 to 2000.0. Default is 0.0.

**FMOD_DSP_SFXREVERB_REVERBDELAY**

Reverb Delay: Late reverberation delay time relative to first reflection in seconds. Ranges from 0.0 to 0.1. Default is 0.04.

**FMOD_DSP_SFXREVERB_DIFFUSION**

Diffusion: Reverberation diffusion (echo density) in percent. Ranges from 0.0 to 100.0. Default is 100.0.

**FMOD_DSP_SFXREVERB_DENSITY**

Density: Reverberation density (modal density) in percent. Ranges from 0.0 to 100.0. Default is 100.0.

**FMOD_DSP_SFXREVERB_HFREFERENCE**
HF Reference : Reference high frequency in Hz. Ranges from 20.0 to 20000.0. Default is 5000.0.

**FMOD_DSP_SFXREVERB_ROOMLF**

Room LF : Room effect low-frequency level in mB. Ranges from -10000.0 to 0.0. Default is 0.0.

**FMOD_DSP_SFXREVERB_LFREFERENCE**

LF Reference : Reference low-frequency in Hz. Ranges from 20.0 to 1000.0. Default is 250.0.

**Remarks**

This is a high quality I3DL2 based reverb. On top of the I3DL2 property set, "Dry Level" is also included to allow the dry mix to be changed.

These properties can be set with presets in **FMOD_REVERB_PRESETS**.

**See Also**

- DSP::setParameter
- DSP::getParameter
- **FMOD_DSP_TYPE**
- System::addDSP
- **FMOD_REVERB_PRESETS**
Firelight Technologies FMOD Ex
FMOD_DSP_TREMOLO

Parameter types for the FMOD_DSP_TYPE_TREMOLO filter.

Enumeration

typedef enum {
    FMOD_DSP_TREMOLO_FREQUENCY,
    FMOD_DSP_TREMOLO_DEPTH,
    FMOD_DSP_TREMOLO_SHAPE,
    FMOD_DSP_TREMOLO_SKEW,
    FMOD_DSP_TREMOLO_DUTY,
    FMOD_DSP_TREMOLO_SQUARE,
    FMOD_DSP_TREMOLO_PHASE,
    FMOD_DSP_TREMOLO_SPREAD
} FMOD_DSP_TREMOLO;

Values

FMOD_DSP_TREMOLO_FREQUENCY
LFO frequency in Hz. 0.1 to 20. Default = 4.

FMOD_DSP_TREMOLO_DEPTH
Tremolo depth. 0 to 1. Default = 0.

FMOD_DSP_TREMOLO_SHAPE
LFO shape morph between triangle and sine. 0 to 1. Default = 0.

FMOD_DSP_TREMOLO_SKEW
Time-skewing of LFO cycle. -1 to 1. Default = 0.

FMOD_DSP_TREMOLO_DUTY
LFO on-time. 0 to 1. Default = 0.5.

FMOD_DSP_TREMOLO_SQUARE
Flatness of the LFO shape. 0 to 1. Default = 0.

*FMOD_DSP_TREMOLO_PHASE*

Instantaneous LFO phase. 0 to 1. Default = 0.

*FMOD_DSP_TREMOLO_SPREAD*

Rotation / auto-pan effect. -1 to 1. Default = 0.

**Remarks**

The tremolo effect varies the amplitude of a sound. Depending on the settings, this unit can produce a tremolo, chopper or auto-pan effect.

The shape of the LFO (low freq. oscillator) can morphed between sine, triangle and sawtooth waves using the *FMOD_DSP_TREMOLO_SHAPE* and *FMOD_DSP_TREMOLO_SKEW* parameters. *FMOD_DSP_TREMOLO_DUTY* and *FMOD_DSP_TREMOLO_SQUARE* are useful for a chopper-type effect where the first controls the on-time duration and second controls the flatness of the envelope. *FMOD_DSP_TREMOLO_SPREAD* varies the LFO phase between channels to get an auto-pan effect. This works best with a sine shape LFO. The LFO can be synchronized using the *FMOD_DSP_TREMOLO_PHASE* parameter which sets its instantaneous phase.

**See Also**

- [DSP::setParameter](#)
- [DSP::getParameter](#)
- [FMOD_DSP_TYPE](#)
Firelight Technologies FMOD Ex
**FMOD_DSP_TYPE**

These definitions can be used for creating FMOD defined special effects or DSP units.

**Enumeration**

```c
typedef enum {
    FMOD_DSP_TYPE_UNKNOWN,
    FMOD_DSP_TYPE_MIXER,
    FMOD_DSP_TYPE_OSCILLATOR,
    FMOD_DSP_TYPE_LOWPASS,
    FMOD_DSP_TYPE_ITLOWPASS,
    FMOD_DSP_TYPE_HIGHPASS,
    FMOD_DSP_TYPE_ECHO,
    FMOD_DSP_TYPE_FLANGE,
    FMOD_DSP_TYPE_DISTORTION,
    FMOD_DSP_TYPE_NORMALIZE,
    FMOD_DSP_TYPE_PARAMEQ,
    FMOD_DSP_TYPE_PITCHSHIFT,
    FMOD_DSP_TYPE_CHORUS,
    FMOD_DSP_TYPE_VSTPLUGIN,
    FMOD_DSP_TYPE_WINAMPPPLUGINS,
    FMOD_DSP_TYPE_ITECHO,
    FMOD_DSP_TYPE_COMPRESSOR,
    FMOD_DSP_TYPE_SFXREVERB,
    FMOD_DSP_TYPE_LOWPASS_SIMPLE,
    FMOD_DSP_TYPE_DELAY,
    FMOD_DSP_TYPE_TREMOLO,
    FMOD_DSP_TYPE_LADSPAPLUGINS,
    FMOD_DSP_TYPE_HIGHPASS_SIMPLE,
    FMOD_DSP_TYPE_HARDWARE
} FMOD_DSP_TYPE;
```

**Values**

**FMOD_DSP_TYPE_UNKNOWN**

This unit was created via a non FMOD plugin so has an unknown purpose.

**FMOD_DSP_TYPE_MIXER**

This unit does nothing but take inputs and mix them together then feed the result
to the soundcard unit.

**FMOD_DSP_TYPE_OSCILLATOR**

This unit generates sine/square/saw/triangle or noise tones.

**FMOD_DSP_TYPE_LOWPASS**

This unit filters sound using a high quality, resonant lowpass filter algorithm but consumes more CPU time.

**FMOD_DSP_TYPE_ITLOWPASS**

This unit filters sound using a resonant lowpass filter algorithm that is used in Impulse Tracker, but with limited cutoff range (0 to 8060hz).

**FMOD_DSP_TYPE_HIGHPASS**

This unit filters sound using a resonant highpass filter algorithm.

**FMOD_DSP_TYPE_ECHO**

This unit produces an echo on the sound and fades out at the desired rate.

**FMOD_DSP_TYPE_FLANGE**

This unit produces a flange effect on the sound.

**FMOD_DSP_TYPE_DISTORTION**

This unit distorts the sound.

**FMOD_DSP_TYPE_NORMALIZE**

This unit normalizes or amplifies the sound to a certain level.

**FMOD_DSP_TYPE_PARAMEQ**

This unit attenuates or amplifies a selected frequency range.

**FMOD_DSP_TYPE_PITCHSHIFT**
This unit bends the pitch of a sound without changing the speed of playback.

**FMOD_DSP_TYPE_CHORUS**

This unit produces a chorus effect on the sound.

**FMOD_DSP_TYPE_VSTPLUGIN**

This unit allows the use of Steinberg VST plugins

**FMOD_DSP_TYPE_WINAMPPPLUGIN**

This unit allows the use of Nullsoft Winamp plugins

**FMOD_DSP_TYPE_ITECHO**

This unit produces an echo on the sound and fades out at the desired rate as is used in Impulse Tracker.

**FMOD_DSP_TYPE_COMPRESSOR**

This unit implements dynamic compression (linked multichannel, wideband)

**FMOD_DSP_TYPE_SFXREVERB**

This unit implements SFX reverb

**FMOD_DSP_TYPE_LOWPASS_SIMPLE**

This unit filters sound using a simple lowpass with no resonance, but has flexible cutoff and is fast.

**FMOD_DSP_TYPE_DELAY**

This unit produces different delays on individual channels of the sound.

**FMOD_DSP_TYPE_TREMOLO**

This unit produces a tremolo / chopper effect on the sound.

**FMOD_DSP_TYPE_LADSPAPLUGIN**
This unit allows the use of LADSPA standard plugins.

*FMOD_DSP_TYPE_HIGHPASS_SIMPLE*

This unit filters sound using a simple highpass with no resonance, but has flexible cutoff and is fast.

*FMOD_DSP_TYPE_HARDWARE*

Offset that platform specific *FMOD_HARDWARE* DSPs will start at.

**Remarks**

To get them to be active, first create the unit, then add it somewhere into the DSP network, either at the front of the network near the soundcard unit to affect the global output (by using System::getDSPHead), or on a single channel (using Channel::getDSPHead).

**See Also**

- System::createDSPByType
Firelight Technologies FMOD Ex
These values describe what state a sound is in after \texttt{FMOD\_NONBLOCKING} has been used to open it.

\textbf{Enumeration}

\begin{verbatim}
typedef enum {
    FMOD\_OPENSTATE\_READY,
    FMOD\_OPENSTATE\_LOADING,
    FMOD\_OPENSTATE\_ERROR,
    FMOD\_OPENSTATE\_CONNECTING,
    FMOD\_OPENSTATE\_BUFFERING,
    FMOD\_OPENSTATE\_SEEKING,
    FMOD\_OPENSTATE\_PLAYING,
    FMOD\_OPENSTATE\_setPosition,
    FMOD\_OPENSTATE\_Max
} FMOD\_OPENSTATE;
\end{verbatim}

\textbf{Values}

\textit{FMOD\_OPENSTATE\_READY}

Opened and ready to play.

\textit{FMOD\_OPENSTATE\_LOADING}

Initial load in progress.

\textit{FMOD\_OPENSTATE\_ERROR}

Failed to open - file not found, out of memory etc. See return value of \texttt{Sound::getOpenState} for what happened.

\textit{FMOD\_OPENSTATE\_CONNECTING}

Connecting to remote host (internet sounds only).

\textit{FMOD\_OPENSTATE\_BUFFERING}
Buffering data.

*FMOD_OPENSTATE_SEEKING*

Seeking to subsound and re-flushing stream buffer.

*FMOD_OPENSTATE_PLAYING*

Ready and playing, but not possible to release at this time without stalling the main thread.

*FMOD_OPENSTATE_SETPOSITION*

Seeking within a stream to a different position.

*FMOD_OPENSTATE_MAX*

Maximum number of open state types.

**Remarks**

With streams, if you are using *FMOD_NONBLOCKING*, note that if the user calls Sound::getSubSound, a stream will go into *FMOD_OPENSTATE_SEEKING* state and sound related commands will return *FMOD_ERR_NOTREADY*.

With streams, if you are using *FMOD_NONBLOCKING*, note that if the user calls Channel::getPosition, a stream will go into *FMOD_OPENSTATE_SETPOSITION* state and sound related commands will return *FMOD_ERR_NOTREADY*.

**See Also**

- [Sound::getOpenState](#)
- [FMOD_MODE](#)
Firelight Technologies FMOD Ex
FMOD_OUTPUTTYPE

These output types are used with System::setOutput / System::getOutput, to choose which output method to use.

Enumeration

typedef enum {
    FMOD_OUTPUTTYPE_AUTODETECT,
    FMOD_OUTPUTTYPE_UNKNOWN,
    FMOD_OUTPUTTYPE_NOSOUND,
    FMOD_OUTPUTTYPE_WAVWRITER,
    FMOD_OUTPUTTYPE_NOSOUND_NRT,
    FMOD_OUTPUTTYPE_WAVWRITER_NRT,
    FMOD_OUTPUTTYPE_DSOUND,
    FMOD_OUTPUTTYPE_WINMM,
    FMOD_OUTPUTTYPE_WASAPI,
    FMOD_OUTPUTTYPE_ASI0,
    FMOD_OUTPUTTYPE_OSS,
    FMOD_OUTPUTTYPE_ALSA,
    FMOD_OUTPUTTYPE_ESD,
    FMOD_OUTPUTTYPE_PULSEAUDIO,
    FMOD_OUTPUTTYPE_COREAUDIO,
    FMOD_OUTPUTTYPE_XBOX360,
    FMOD_OUTPUTTYPE_PSP,
    FMOD_OUTPUTTYPE_PS3,
    FMOD_OUTPUTTYPE_NGP,
    FMOD_OUTPUTTYPE_WII,
    FMOD_OUTPUTTYPE_3DS,
    FMOD_OUTPUTTYPE_AUDIOTRACK,
    FMOD_OUTPUTTYPE_OPENSL,
    FMOD_OUTPUTTYPE_NACL,
    FMOD_OUTPUTTYPE_WIIU,
    FMOD_OUTPUTTYPE_WIIU_EMIT,
    FMOD_OUTPUTTYPE_MAX
} FMOD_OUTPUTTYPE;

Values

FMOD_OUTPUTTYPE_AUTODETECT

Picks the best output mode for the platform. This is the default.
**FMOD_OUTPUTTYPE_UNKNOWN**

All - 3rd party plugin, unknown. This is for use with `System::getOutput` only.

**FMOD_OUTPUTTYPE_NOSOUND**

All - All calls in this mode succeed but make no sound.

**FMOD_OUTPUTTYPE_WAVWRITER**

All - Writes output to fmodoutput.wav by default. Use the 'extradriverrdata' parameter in `System::init`, by simply passing the filename as a string, to set the wav filename.

**FMOD_OUTPUTTYPE_NOSOUND_NRT**

All - Non-realtime version of `FMOD_OUTPUTTYPE_NOSOUND`. User can drive mixer with `System::update` at whatever rate they want.

**FMOD_OUTPUTTYPE_WAVWRITER_NRT**

All - Non-realtime version of `FMOD_OUTPUTTYPE_WAVWRITER`. User can drive mixer with `System::update` at whatever rate they want.

**FMOD_OUTPUTTYPE_DSOUND**

Win32/Win64 - DirectSound output. (Default on Windows XP and below)

**FMOD_OUTPUTTYPE_WINMM**

Win32/Win64 - Windows Multimedia output.

**FMOD_OUTPUTTYPE_WASAPI**

Win32 - Windows Audio Session API. (Default on Windows Vista and above)

**FMOD_OUTPUTTYPE_ASIO**

Win32 - Low latency ASIO 2.0 driver.

**FMOD_OUTPUTTYPE_OSS**
Linux/Linux64 - Open Sound System output. (Default on Linux, third preference)

FMOD_OUTPUTTYPE_ALSA

Linux/Linux64 - Advanced Linux Sound Architecture output. (Default on Linux, second preference if available)

FMOD_OUTPUTTYPE_ESD

Linux/Linux64 - Enlightenment Sound Daemon output.

FMOD_OUTPUTTYPE_PULSEAUDIO

Linux/Linux64 - PulseAudio output. (Default on Linux, first preference if available)

FMOD_OUTPUTTYPE_COREAUDIO

Mac - Macintosh CoreAudio output. (Default on Mac)

FMOD_OUTPUTTYPE_XBOX360

Xbox 360 - Native Xbox360 output. (Default on Xbox 360)

FMOD_OUTPUTTYPE_PSP

PSP - Native PSP output. (Default on PSP)

FMOD_OUTPUTTYPE_PS3

PS3 - Native PS3 output. (Default on PS3)

FMOD_OUTPUTTYPE_NGP

NGP - Native NGP output. (Default on NGP)

FMOD_OUTPUTTYPE_WII

Wii - Native Wii output. (Default on Wii)
**FMOD_OUTPUTTYPE_3DS**

3DS - Native 3DS output (Default on 3DS)

**FMOD_OUTPUTTYPE_AUDIOTRACK**

Android - Java Audio Track output. (Default on Android 2.2 and below)

**FMOD_OUTPUTTYPE_OPENSL**

Android - OpenSL ES output. (Default on Android 2.3 and above)

**FMOD_OUTPUTTYPE_NACL**

Native Client - Native Client output. (Default on Native Client)

**FMOD_OUTPUTTYPE_WIIU**

Wii U - Native Wii U output. (Default on Wii U)

**FMOD_OUTPUTTYPE_AŚOUND**

BlackBerry - Native BlackBerry asound output. (Default on BlackBerry)

**FMOD_OUTPUTTYPE_MAX**

Maximum number of output types supported.

**Remarks**

To pass information to the driver when initializing fmod use the extradrivert data parameter in `System::init` for the following reasons.

- **FMOD_OUTPUTTYPE_WAVWRITER** - extradrivert data is a pointer to a char *
  * filename that the wav writer will output to.
- **FMOD_OUTPUTTYPE_WAVWRITER_NRT** - extradrivert data is a pointer to a char *
  * filename that the wav writer will output to.
- **FMOD_OUTPUTTYPE_DSOUND** - extradrivert data is a pointer to a HWND so that FMOD can set the focus on the audio for a particular window.
- FMOD_OUTPUTTYPE_PS3 - extradriverdata is a pointer to a FMOD_PS3_EXTRADRIVERDATA struct. This can be found in fmodps3.h.
- FMOD_OUTPUTTYPE_GC - extradriverdata is a pointer to a FMOD_GC_INFO struct. This can be found in fmodgc.h.
- FMOD_OUTPUTTYPE_WII - extradriverdata is a pointer to a FMOD_WII_INFO struct. This can be found in fmodwii.h.
- FMOD_OUTPUTTYPE_ALSA - extradriverdata is a pointer to a FMOD_LINUX_EXTRADRIVERDATA struct. This can be found in fmodlinux.h.

Currently these are the only FMOD drivers that take extra information. Other unknown plugins may have different requirements.

Note! If FMOD_OUTPUTTYPE_WAVWRITER_NRT or FMOD_OUTPUTTYPE_NOSOUND_NRT are used, and if the System::update function is being called very quickly (ie for a non realtime decode) it may be being called too quickly for the FMOD streamer thread to respond to. The result will be a skipping/stuttering output in the captured audio.

To remedy this, disable the FMOD Ex streamer thread, and use FMOD_INIT_STREAM_FROM_UPDATE to avoid skipping in the output stream, as it will lock the mixer and the streamer together in the same thread.

See Also

- System::setOutput
- System::getOutput
- System::setSoftwareFormat
- System::getSoftwareFormat
- System::init
- System::update
- FMOD_INITFLAGS
Firelight Technologies FMOD Ex
FMOD_PLUGINTYPE

These are plugin types defined for use with the System::getNumPlugins, System::getPluginInfo and System::unloadPlugin functions.

Enumeration

typedef enum {
    FMOD_PLUGINTYPE_OUTPUT,
    FMOD_PLUGINTYPE_CODEC,
    FMOD_PLUGINTYPE_DSP,
    FMOD_PLUGINTYPE_MAX
} FMOD_PLUGINTYPE;

Values

**FMOD_PLUGINTYPE_OUTPUT**

The plugin type is an output module. FMOD mixed audio will play through one of these devices

**FMOD_PLUGINTYPE_CODEC**

The plugin type is a file format codec. FMOD will use these codecs to load file formats for playback.

**FMOD_PLUGINTYPE_DSP**

The plugin type is a DSP unit. FMOD will use these plugins as part of its DSP network to apply effects to output or generate sound in realtime.

**FMOD_PLUGINTYPE_MAX**

Maximum number of plugin types supported.

See Also
- System::getNumPlugins
- System::getPluginInfo
- System::unloadPlugin

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
FMOD_PSP_OUTPUTMODE

Enumeration

typedef enum {
    FMOD_PSP_OUTPUTMODE_SIMPLEAUDIO,
    FMOD_PSP_OUTPUTMODE_LIBWAVE
} FMOD_PSP_OUTPUTMODE;

Values

FMOD_PSP_OUTPUTMODE_SIMPLEAUDIO
Simple Audio Output (Default)

FMOD_PSP_OUTPUTMODE_LIBWAVE
libWave

See Also

- FMOD_PSP_ChangeOutputMode
Firelight Technologies FMOD Ex
FMOD_RESULT

error codes. Returned from every function.

Enumeration

typedef enum {
    FMOD_OK,
    FMOD_ERR_ALREADYLOCKED,
    FMOD_ERR_BADCOMMAND,
    FMOD_ERR_CDDA_DRIVERS,
    FMOD_ERR_CDDA_INIT,
    FMOD_ERR_CDDA_INVALID_DEVICE,
    FMOD_ERR_CDDA_NOAUDIO,
    FMOD_ERR_CDDA_NODEVICES,
    FMOD_ERR_CDDA_NODISC,
    FMOD_ERR_CDDA_READ,
    FMOD_ERR_CHANNEL_ALLOC,
    FMOD_ERR_CHANNEL_STOLEN,
    FMOD_ERR_COM,
    FMOD_ERR_DMA,
    FMOD_ERR_DSP_CONNECTION,
    FMOD_ERR_DSP_FORMAT,
    FMOD_ERR_DSP_NOTFOUND,
    FMOD_ERR_DSP_RUNNING,
    FMOD_ERR_DSP_TOOMANYCONNECTIONS,
    FMOD_ERR_FILE_BAD,
    FMOD_ERR_FILE_CouldNOTSEEK,
    FMOD_ERR_FILE_DISJECTED,
    FMOD_ERR_FILE_EOF,
    FMOD_ERR_FILE_NOTFOUND,
    FMOD_ERR_FILE_UNWANTED,
    FMOD_ERR_FORMAT,
    FMOD_ERR_HTTP,
    FMOD_ERR_HTTP_ACCESS,
    FMOD_ERR_HTTP_PROXY_AUTH,
    FMOD_ERR_HTTP_SERVER_ERROR,
    FMOD_ERR_HTTP_TIMEOUT,
    FMOD_ERR_INITIALIZATION,
    FMOD_ERR_INITIALIZED,
    FMOD_ERR_INTERNAL,
    FMOD_ERR_INVALID_ADDRESS,
    FMOD_ERR_INVALID_FLOAT,
    FMOD_ERR_INVALID_HANDLE,
    FMOD_ERR_INVALID_PARAM,
}
FMOD_ERR_INVALID_POSITION,
FMOD_ERR_INVALID_SPEAKER,
FMOD_ERR_INVALID_SYNCPOINT,
FMOD_ERR_INVALID_VECTOR,
FMOD_ERR_MAXAUDIBLE,
FMOD_ERR_MEMORY,
FMOD_ERR_MEMORY_CANTPOINT,
FMOD_ERR_MEMORY_SRAM,
FMOD_ERR_NEEDS2D,
FMOD_ERR_NEEDS3D,
FMOD_ERR_NEEDSHARDWARE,
FMOD_ERR_NEEDSSOFTWARE,
FMOD_ERR_NET_CONNECT,
FMOD_ERR_NET_SOCKET_ERROR,
FMOD_ERR_NET_URL,
FMOD_ERR_NET_WOULD_BLOCK,
FMOD_ERR_NOTREADY,
FMOD_ERR_OUTPUT_ALLOCATED,
FMOD_ERR_OUTPUT_CREATEBUFFER,
FMOD_ERR_OUTPUT_DRIVERCALL,
FMOD_ERR_OUTPUT_ENUMERATION,
FMOD_ERR_OUTPUT_FORMAT,
FMOD_ERR_OUTPUT_INIT,
FMOD_ERR_OUTPUT_NOHARDWARE,
FMOD_ERR_OUTPUT_NOSOFTWARE,
FMOD_ERR_PAN,
FMOD_ERR_PLUGIN,
FMOD_ERR_PLUGIN_INSTANCES,
FMOD_ERR_PLUGIN_MISSING,
FMOD_ERR_PLUGIN_RESOURCE,
FMOD_ERR_PRELOADED,
FMOD_ERR_PROGRAMMERSOUND,
FMOD_ERR_RECORD,
FMOD_ERR_REVERB_INSTANCE,
FMOD_ERR_SUBSOUND_ALLOCATED,
FMOD_ERR_SUBSOUND_CANTMOVE,
FMOD_ERR_SUBSOUND_MODE,
FMOD_ERR_SUBSOUNDS,
FMOD_ERR_TAGNOTFOUND,
FMOD_ERR_TOOMANYCHANNELS,
FMOD_ERR_UNIMPLEMENTED,
FMOD_ERR_UNINITIALIZED,
FMOD_ERR_UNSUPPORTED,
FMOD_ERR_UPDATE,
FMOD_ERR_VERSION,
FMOD_ERR_EVENT_FAILED,
FMOD_ERR_EVENT_INFOONLY,
FMOD_ERR_EVENT_INTERNAL,
FMOD_ERR_EVENT_MAXSTREAMS,
FMOD_ERR_EVENT_MISMATCH,
FMOD_ERR_EVENT_NAMECONFLICT,
FMOD_ERR_EVENT_NOTFOUND,
FMOD_ERR_EVENT_NEEDSSIMPLE,
FMOD_ERR_EVENT_GUIDCONFLICT,
FMOD_ERR_EVENT_ALREADY_LOADED,
FMOD_ERR_MUSIC_UNINITIALIZED,
FMOD_ERR_MUSIC_NOTFOUND,
FMOD_ERR_MUSIC_NOCALLBACK
} FMOD_RESULT;

Values

FMOD_OK
No errors.

FMOD_ERR_ALREADYLOCKED
Tried to call lock a second time before unlock was called.

FMOD_ERR_BADCOMMAND
Tried to call a function on a data type that does not allow this type of functionality (ie calling Sound::lock on a streaming sound).

FMOD_ERR_CDDA_DRIVERS
Neither NTSCSI nor ASPI could be initialised.

FMOD_ERR_CDDA_INIT
An error occurred while initialising the CDDA subsystem.

FMOD_ERR_CDDA_INVALID_DEVICE
Couldn't find the specified device.

FMOD_ERR_CDDA_NOAUDIO
No audio tracks on the specified disc.

FMOD_ERR_CDDA_NODEVICES
No CD/DVD devices were found.

`FMOD_ERR_CDDA_NODISC`
No disc present in the specified drive.

`FMOD_ERR_CDDA_READ`
A CDDA read error occurred.

`FMOD_ERR_CHANNEL_ALLOC`
Error trying to allocate a channel.

`FMOD_ERR_CHANNEL_STOLEN`
The specified channel has been reused to play another sound.

`FMOD_ERR_COM`
A Win32 COM related error occurred. COM failed to initialize or a QueryInterface failed meaning a Windows codec or driver was not installed properly.

`FMOD_ERR_DMA`
DMA Failure. See debug output for more information.

`FMOD_ERR_DSP_CONNECTION`
DSP connection error. Connection possibly caused a cyclic dependancy. Or tried to connect a tree too many units deep (more than 128).

`FMOD_ERR_DSP_FORMAT`
DSP Format error. A DSP unit may have attempted to connect to this network with the wrong format.

`FMOD_ERR_DSP_NOTFOUND`
DSP connection error. Couldn't find the DSP unit specified.
**FMOD_ERR_DSP_RUNNING**

DSP error. Cannot perform this operation while the network is in the middle of running. This will most likely happen if a connection or disconnection is attempted in a DSP callback.

**FMOD_ERR_DSP_TOOMANYCONNECTIONS**

DSP connection error. The unit being connected to or disconnected should only have 1 input or output.

**FMOD_ERR_FILE_BAD**

Error loading file.

**FMOD_ERR_FILE_COULDNOTSEEK**

Couldn't perform seek operation. This is a limitation of the medium (ie netstreams) or the file format.

**FMOD_ERR_FILE_DISKEJECTED**

Media was ejected while reading.

**FMOD_ERR_FILE_EOF**

End of file unexpectedly reached while trying to read essential data (truncated data?).

**FMOD_ERR_FILE_NOTFOUND**

File not found.

**FMOD_ERR_FILE_UNWANTED**

Unwanted file access occurred.

**FMOD_ERR_FORMAT**

Unsupported file or audio format.
**FMOD_ERR_HTTP**
A HTTP error occurred. This is a catch-all for HTTP errors not listed elsewhere.

**FMOD_ERR_HTTP_ACCESS**
The specified resource requires authentication or is forbidden.

**FMOD_ERR_HTTP_PROXY_AUTH**
Proxy authentication is required to access the specified resource.

**FMOD_ERR_HTTP_SERVER_ERROR**
A HTTP server error occurred.

**FMOD_ERR_HTTP_TIMEOUT**
The HTTP request timed out.

**FMOD_ERR_INITIALIZATION**
FMOD was not initialized correctly to support this function.

**FMOD_ERR_INITIALIZED**
Cannot call this command after System::init.

**FMOD_ERR_INTERNAL**
An error occurred that wasn't supposed to. Contact support.

**FMOD_ERR_INVALID_ADDRESS**
On Xbox 360, this memory address passed to FMOD must be physical, (ie allocated with XPhysicalAlloc.)

**FMOD_ERR_INVALID_FLOAT**
Value passed in was a NaN, Inf or denormalized float.
**FMOD_ERR_INVALID_HANDLE**
An invalid object handle was used.

**FMOD_ERR_INVALID_PARAM**
An invalid parameter was passed to this function.

**FMOD_ERR_INVALID_POSITION**
An invalid seek position was passed to this function.

**FMOD_ERR_INVALID_SPEAKER**
An invalid speaker was passed to this function based on the current speaker mode.

**FMOD_ERR_INVALID_SYNCPOINT**
The syncpoint did not come from this sound handle.

**FMOD_ERR_INVALID_VECTOR**
The vectors passed in are not unit length, or perpendicular.

**FMOD_ERR_MAXAUDIBLE**
Reached maximum audible playback count for this sound's soundgroup.

**FMOD_ERR_MEMORY**
Not enough memory or resources.

**FMOD_ERR_MEMORY_CANTPOINT**
Can't use **FMOD_OPENMEMORY_POINT** on non PCM source data, or non mp3/xma/adpcm data if **FMOD_CREATECOMPRESSESAMPLE** was used.

**FMOD_ERR_MEMORY_SRAM**
Not enough memory or resources on console sound ram.
**FMOD_ERR_NEEDS2D**

Tried to call a command on a 3d sound when the command was meant for 2d sound.

**FMOD_ERR_NEEDS3D**

Tried to call a command on a 2d sound when the command was meant for 3d sound.

**FMOD_ERR_NEEDSHARDWARE**

Tried to use a feature that requires hardware support. (ie trying to play a GCADPCM compressed sound in software on Wii).

**FMOD_ERR_NEEDSSOFTWARE**

Tried to use a feature that requires the software engine. Software engine has either been turned off, or command was executed on a hardware channel which does not support this feature.

**FMOD_ERR_NET_CONNECT**

Couldn't connect to the specified host.

**FMOD_ERR_NET_SOCKET_ERROR**

A socket error occurred. This is a catch-all for socket-related errors not listed elsewhere.

**FMOD_ERR_NET_URL**

The specified URL couldn't be resolved.

**FMOD_ERR_NET_WOULD_BLOCK**

Operation on a non-blocking socket could not complete immediately.

**FMOD_ERR_NOTREADY**

Operation could not be performed because specified sound/DSP connection is
not ready.

**FMOD_ERR_OUTPUT_ALLOCATED**

Error initializing output device, but more specifically, the output device is already in use and cannot be reused.

**FMOD_ERR_OUTPUT_CREATEBUFFER**

Error creating hardware sound buffer.

**FMOD_ERR_OUTPUT_DRIVERCALL**

A call to a standard soundcard driver failed, which could possibly mean a bug in the driver or resources were missing or exhausted.

**FMOD_ERR_OUTPUT_ENUMERATION**

Error enumerating the available driver list. List may be inconsistent due to a recent device addition or removal.

**FMOD_ERR_OUTPUT_FORMAT**

Soundcard does not support the minimum features needed for this soundsystem (16bit stereo output).

**FMOD_ERR_OUTPUT_INIT**

Error initializing output device.

**FMOD_ERR_OUTPUT_NOHARDWARE**

FMOD_HARDWARE was specified but the sound card does not have the resources necessary to play it.

**FMOD_ERR_OUTPUT_NOSOFTWARE**

Attempted to create a software sound but no software channels were specified in System::init.

**FMOD_ERR_PAN**
Panning only works with mono or stereo sound sources.

**FMOD_ERR_PLUGIN**

An unspecified error has been returned from a 3rd party plugin.

**FMOD_ERR_PLUGIN_INSTANCES**

The number of allowed instances of a plugin has been exceeded.

**FMOD_ERR_PLUGIN_MISSING**

A requested output, dsp unit type or codec was not available.

**FMOD_ERR_PLUGIN_RESOURCE**

A resource that the plugin requires cannot be found. (ie the DLS file for MIDI playback)

**FMOD_ERR_PRELOADED**

The specified sound is still in use by the event system, call EventSystem::unloadFSB before trying to release it.

**FMOD_ERR_PROGRAMMERSOUND**

The specified sound is still in use by the event system, wait for the event which is using it finish with it.

**FMOD_ERR_RECORD**

An error occurred trying to initialize the recording device.

**FMOD_ERR_REVERB_INSTANCE**

Specified instance in FMOD_REVERB_PROPERTIES couldn't be set. Most likely because it is an invalid instance number or the reverb doesn't exist.

**FMOD_ERR_SUBSOUND_ALLOCATED**

This subsound is already being used by another sound, you cannot have more
than one parent to a sound. Null out the other parent's entry first.

**FMOD_ERR_SUBSOUND_CANTMOVE**

Shared subsounds cannot be replaced or moved from their parent stream, such as when the parent stream is an FSB file.

**FMOD_ERR_SUBSOUND_MODE**

The subsound's mode bits do not match with the parent sound's mode bits. See documentation for function that it was called with.

**FMOD_ERR_SUBSOUNDS**

The error occurred because the sound referenced contains subsounds when it shouldn't have, or it doesn't contain subsounds when it should have. The operation may also not be able to be performed on a parent sound, or a parent sound was played without setting up a sentence first.

**FMOD_ERR_TAGNOTFOUND**

The specified tag could not be found or there are no tags.

**FMOD_ERR_TOOMANYCHANNELS**

The sound created exceeds the allowable input channel count. This can be increased using the maxinputchannels parameter in System::setSoftwareFormat.

**FMOD_ERR_UNIMPLEMENTED**

Something in FMOD hasn't been implemented when it should be! contact support!

**FMOD_ERR_UNINITIALIZED**

This command failed because System::init or System::setDriver was not called.

**FMOD_ERR_UNSUPPORTED**

A command issued was not supported by this object. Possibly a plugin without certain callbacks specified.
**FMOD_ERR_UPDATE**

An error caused by System::update occurred.

**FMOD_ERR_VERSION**

The version number of this file format is not supported.

**FMOD_ERR_EVENT_FAILED**

An Event failed to be retrieved, most likely due to 'just fail' being specified as the max playbacks behavior.

**FMOD_ERR_EVENT_INFOONLY**

Can't execute this command on an EVENT_INFOONLY event.

**FMOD_ERR_EVENT_INTERNAL**

An error occurred that wasn't supposed to. See debug log for reason.

**FMOD_ERR_EVENT_MAXSTREAMS**

Event failed because 'Max streams' was hit when **FMOD_EVENT_INIT_FAIL_ON_MAXSTREAMS** was specified.

**FMOD_ERR_EVENT_MISMATCH**

FSB mismatches the FEV it was compiled with, the stream/sample mode it was meant to be created with was different, or the FEV was built for a different platform.

**FMOD_ERR_EVENT_NAMECONFLICT**

A category with the same name already exists.

**FMOD_ERR_EVENT_NOTFOUND**

The requested event, event group, event category or event property could not be found.
FMOD_ERR_EVENT_NEEDSSIMPLE

Tried to call a function on a complex event that's only supported by simple events.

FMOD_ERR_EVENT_GUIDCONFLICT

An event with the same GUID already exists.

FMOD_ERR_EVENT_ALREADY_LOADED

The specified project or bank has already been loaded. Having multiple copies of the same project loaded simultaneously is forbidden.

FMOD_ERR_MUSIC_UNINITIALIZED

Music system is not initialized probably because no music data is loaded.

FMOD_ERR_MUSIC_NOTFOUND

The requested music entity could not be found.

FMOD_ERR_MUSIC_NOCALLBACK

The music callback is required, but it has not been set.
Firelight Technologies FMOD Ex
FMOD_SOUNDGROUP_BEHAVIOR

These flags are used with SoundGroup::setMaxAudibleBehavior to determine what happens when more sounds are played than are specified with SoundGroup::setMaxAudible.

Enumeration

typedef enum {
    FMOD_SOUNDGROUP_BEHAVIOR_FAIL,
    FMOD_SOUNDGROUP_BEHAVIOR_MUTE,
    FMOD_SOUNDGROUP_BEHAVIOR_STEALLOWEST,
    FMOD_SOUNDGROUP_BEHAVIOR_MAX
} FMOD_SOUNDGROUP_BEHAVIOR;

Values

FMOD_SOUNDGROUP_BEHAVIOR_FAIL

Any sound played that puts the sound count over the SoundGroup::setMaxAudible setting, will simply fail during System::playSound.

FMOD_SOUNDGROUP_BEHAVIOR_MUTE

Any sound played that puts the sound count over the SoundGroup::setMaxAudible setting, will be silent, then if another sound in the group stops the sound that was silent before becomes audible again.

FMOD_SOUNDGROUP_BEHAVIOR_STEALLOWEST

Any sound played that puts the sound count over the SoundGroup::setMaxAudible setting, will steal the quietest / least important sound playing in the group.

FMOD_SOUNDGROUP_BEHAVIOR_MAX

Maximum number of open state types.
Remarks

When using `FMOD_SOUNDPACK_BEHAVIOR_MUTE`, `SoundGroup::setMuteFadeSpeed` can be used to stop a sudden transition. Instead, the time specified will be used to cross fade between the sounds that go silent and the ones that become audible.

See Also

- `SoundGroup::setMaxAudibleBehavior`
- `SoundGroup::getMaxAudibleBehavior`
- `SoundGroup::setMaxAudible`
- `SoundGroup::getMaxAudible`
- `SoundGroup::setMuteFadeSpeed`
- `SoundGroup::getMuteFadeSpeed`
FMOD_SOUND_FORMAT

These definitions describe the native format of the hardware or software buffer that will be used.

Enumeration

typedef enum {
    FMOD_SOUND_FORMAT_NONE,
    FMOD_SOUND_FORMAT_PCM8,
    FMOD_SOUND_FORMAT_PCM16,
    FMOD_SOUND_FORMAT_PCM24,
    FMOD_SOUND_FORMAT_PCM32,
    FMOD_SOUND_FORMAT_PCMFLOAT,
    FMOD_SOUND_FORMAT_GCADPCM,
    FMOD_SOUND_FORMAT_IMAADPCM,
    FMOD_SOUND_FORMAT_VAG,
    FMOD_SOUND_FORMAT_HEVAG,
    FMOD_SOUND_FORMAT_XMA,
    FMOD_SOUND_FORMAT_MPEG,
    FMOD_SOUND_FORMAT_CELT,
    FMOD_SOUND_FORMAT_AT9,
    FMOD_SOUND_FORMAT_XWMA,
    FMOD_SOUND_FORMAT_VORBIS,
    FMOD_SOUND_FORMAT_MAX
} FMOD_SOUND_FORMAT;

Values

FMOD_SOUND_FORMAT_NONE

Uninitialized / unknown.

FMOD_SOUND_FORMAT_PCM8

8bit integer PCM data.

FMOD_SOUND_FORMAT_PCM16

16bit integer PCM data.

FMOD_SOUND_FORMAT_PCM24
24bit integer PCM data.

*FMOD_SOUND_FORMAT_PCM32*

32bit integer PCM data.

*FMOD_SOUND_FORMAT_PCMFLOAT*

32bit floating point PCM data.

*FMOD_SOUND_FORMAT_GCADPCM*

Compressed Nintendo 3DS/Wii DSP data.

*FMOD_SOUND_FORMAT_IMAADPCM*

Compressed IMA ADPCM data.

*FMOD_SOUND_FORMAT_VAG*

Compressed PlayStation Portable ADPCM data.

*FMOD_SOUND_FORMAT_HEVAG*

Compressed PSVita ADPCM data.

*FMOD_SOUND_FORMAT_XMA*

Compressed Xbox360 XMA data.

*FMOD_SOUND_FORMAT_MPEG*

Compressed MPEG layer 2 or 3 data.

*FMOD_SOUND_FORMAT_CELT*

Compressed CELT data.

*FMOD_SOUND_FORMAT_AT9*

Compressed PSVita ATRAC9 data.
**FMOD_SOUND_FORMAT_XWMA**

Compressed Xbox360 xWMA data.

**FMOD_SOUND_FORMAT_VORBIS**

Compressed Vorbis data.

**FMOD_SOUND_FORMAT_MAX**

Maximum number of sound formats supported.

**Remarks**

This is the format the native hardware or software buffer will be or is created in.

**See Also**

- [System::createSound](#)
- [Sound::getFormat](#)
Firelight Technologies FMOD Ex
**FMOD_SOUND_TYPE**

These definitions describe the type of song being played.

**Enumeration**

```c
typedef enum {
    FMOD_SOUND_TYPE_UNKNOWN,
    FMOD_SOUND_TYPE_AIFF,
    FMOD_SOUND_TYPEASF,
    FMOD_SOUND_TYPE_AT3,
    FMOD_SOUND_TYPE_CDDA,
    FMOD_SOUND_TYPE_DLS,
    FMOD_SOUND_TYPE_FLAC,
    FMOD_SOUND_TYPE_FSB,
    FMOD_SOUND_TYPE_GCADPCM,
    FMOD_SOUND_TYPE_IT,
    FMOD_SOUND_TYPE_MIDI,
    FMOD_SOUND_TYPE_MOD,
    FMOD_SOUND_TYPE_MPEG,
    FMOD_SOUND_TYPE_OGGVORBIS,
    FMOD_SOUND_TYPE_PLAYLIST,
    FMOD_SOUND_TYPE_RAW,
    FMOD_SOUND_TYPE_S3M,
    FMOD_SOUND_TYPE_SF2,
    FMOD_SOUND_TYPE_USER,
    FMOD_SOUND_TYPE_WAV,
    FMOD_SOUND_TYPE_XM,
    FMOD_SOUND_TYPE_XMA,
    FMOD_SOUND_TYPE_VAG,
    FMOD_SOUND_TYPE_AUDIOQUEUE,
    FMOD_SOUND_TYPE_XWMA,
    FMOD_SOUND_TYPE_BCWAV,
    FMOD_SOUND_TYPE_AT9,
    FMOD_SOUND_TYPE_VORBIS,
    FMOD_SOUND_TYPE_MEDIA_FOUNDATION,
    FMOD_SOUND_TYPE_MAX
} FMOD_SOUND_TYPE;
```

**Values**

*FMOD_SOUND_TYPE_UNKNOWN*

3rd party / unknown plugin format.
*FMOD_SOUND_TYPE_AIFF*

AIFF.

*FMOD_SOUND_TYPE_ASF*

Microsoft Advanced Systems Format (ie WMA/ASF/WMV).

*FMOD_SOUND_TYPE_AT3*

Sony ATRAC 3 format

*FMOD_SOUND_TYPE_CDDA*

Digital CD audio.

*FMOD_SOUND_TYPE_DLS*

Sound font / downloadable sound bank.

*FMOD_SOUND_TYPE_FLAC*

FLAC lossless codec.

*FMOD_SOUND_TYPE_FSB*

FMOD Sample Bank.

*FMOD_SOUND_TYPE_GCADPCM*

Nintendo GameCube/Wii ADPCM

*FMOD_SOUND_TYPE_IT*

Impulse Tracker.

*FMOD_SOUND_TYPE_MIDI*

MIDI. extracodecdatal is a pointer to an FMOD_MIDI_EXTRACODECDATA structure.
**FMOD_SOUND_TYPE_MOD**
Protracker / Fasttracker MOD.

**FMOD_SOUND_TYPE_MPEG**
MP2/MP3 MPEG.

**FMOD_SOUND_TYPE_OGGVORBIS**
Ogg vorbis.

**FMOD_SOUND_TYPE_PLAYLIST**
Information only from ASX/PLS/M3U/WAX playlists

**FMOD_SOUND_TYPE_RAW**
Raw PCM data.

**FMOD_SOUND_TYPE_S3M**
ScreamTracker 3.

**FMOD_SOUND_TYPE_SF2**
Sound font 2 format.

**FMOD_SOUND_TYPE_USER**
User created sound.

**FMOD_SOUND_TYPE_WAV**
Microsoft WAV.

**FMOD_SOUND_TYPE_XM**
FastTracker 2 XM.

**FMOD_SOUND_TYPE_XMA**

Xbox360 XMA

FMOD_SOUND_TYPE_VAG

PlayStation Portable ADPCM VAG format.

FMOD_SOUND_TYPE_AUDIOQUEUE

iPhone hardware decoder, supports AAC, ALAC and MP3. extracodecdata is a pointer to an FMOD_AUDIOQUEUE_EXTRACODECDATA structure.

FMOD_SOUND_TYPE_XWMA

Xbox360 XWMA

FMOD_SOUND_TYPE_BCWAV

3DS BCWAV container format for DSP ADPCM and PCM

FMOD_SOUND_TYPE_AT9

NGP ATRAC 9 format

FMOD_SOUND_TYPE_VORBIS

Raw vorbis

FMOD_SOUND_TYPE_MEDIA_FOUNDATION

Microsoft Media Foundation wrappers, supports ASF/WMA

FMOD_SOUND_TYPE_MAX

Maximum number of sound types supported.

See Also

- Sound::getFormat
Firelight Technologies FMOD Ex
These are speaker types defined for use with the `Channel::setSpeakerLevels` command. It can also be used for speaker placement in the `System::set3DSpeakerPosition` command.

**Enumeration**

```c
typedef enum {
    FMOD_SPEAKER_FRONT_LEFT,
    FMOD_SPEAKER_FRONT_RIGHT,
    FMOD_SPEAKER_FRONT_CENTER,
    FMOD_SPEAKER_LOW_FREQUENCY,
    FMOD_SPEAKER_BACK_LEFT,
    FMOD_SPEAKER_BACK_RIGHT,
    FMOD_SPEAKER_SIDE_LEFT,
    FMOD_SPEAKER_SIDE_RIGHT,
    FMOD_SPEAKER_MAX,
    FMOD_SPEAKER_MONO,
    FMOD_SPEAKER_NULL,
    FMOD_SPEAKER_SBL,
    FMOD_SPEAKER_SBR
} FMOD_SPEAKER;
```

**Values**

- `FMOD_SPEAKER_FRONT_LEFT`
- `FMOD_SPEAKER_FRONT_RIGHT`
- `FMOD_SPEAKER_FRONT_CENTER`
- `FMOD_SPEAKER_LOW_FREQUENCY`
- `FMOD_SPEAKER_BACK_LEFT`
- `FMOD_SPEAKER_BACK_RIGHT`
- `FMOD_SPEAKER_SIDE_LEFT`
- `FMOD_SPEAKER_SIDE_RIGHT`
**FMOD_SPEAKER_MAX**

Maximum number of speaker types supported.

**FMOD_SPEAKER_MONO**

For use with `FMOD_SPEAKERMODE_MONO` and `Channel::SetSpeakerLevels`. Mapped to same value as `FMOD_SPEAKER_FRONT_LEFT`.

**FMOD_SPEAKER_NULL**

A non-sounder. Use this with ASIO mapping to ignore a speaker.

**FMOD_SPEAKER_SBL**

For use with `FMOD_SPEAKERMODE_7POINT1` on PS3 where the extra speakers are surround back inside of side speakers.

**FMOD_SPEAKER_SBR**

For use with `FMOD_SPEAKERMODE_7POINT1` on PS3 where the extra speakers are surround back inside of side speakers.

**Remarks**

If you are using `FMOD_SPEAKERMODE_RAW` and speaker assignments are meaningless, just cast a raw integer value to this type. For example `(FMOD_SPEAKER)7` would use the 7th speaker (also the same as `FMOD_SPEAKER_SIDE_RIGHT`). Values higher than this can be used if an output system has more than 8 speaker types / output channels. 15 is the current maximum.

NOTE: On Playstation 3 in 7.1, the extra 2 speakers are not side left/side right, they are 'surround back left'/surround back right' which locate the speakers behind the listener instead of to the sides like on PC. `FMOD_SPEAKER_SBL/FMOD_SPEAKER_SBR` are provided to make it clearer what speaker is being addressed on that platform.
See Also

- `FMOD_SPEAKERMODE`
- `Channel::setSpeakerLevels`
- `Channel::getSpeakerLevels`
- `System::set3DSpeakerPosition`
- `System::get3DSpeakerPosition`
Firelight Technologies FMOD Ex
**FMOD_SPEAKERMAPTYPE**

When creating a multichannel sound, FMOD will pan them to their default speaker locations, for example a 6 channel sound will default to one channel per 5.1 output speaker.

Another example is a stereo sound. It will default to left = front left, right = front right.

This is for sounds that are not 'default'. For example you might have a sound that is 6 channels but actually made up of 3 stereo pairs, that should all be located in front left, front right only.

**Enumeration**

```c
typedef enum {
    FMOD_SPEAKERMAPTYPE_DEFAULT,
    FMOD_SPEAKERMAPTYPE_ALLMONO,
    FMOD_SPEAKERMAPTYPE_ALLSTEREO,
    FMOD_SPEAKERMAPTYPE_51_PROTOOLS
} FMOD_SPEAKERMAPTYPE;
```

**Values**

**FMOD_SPEAKERMAPTYPE_DEFAULT**

This is the default, and just means FMOD decides which speakers it puts the source channels.

**FMOD_SPEAKERMAPTYPE_ALLMONO**

This means the sound is made up of all mono sounds. All voices will be panned to the front center by default in this case.

**FMOD_SPEAKERMAPTYPE_ALLSTEREO**

This means the sound is made up of all stereo sounds. All voices will be panned to front left and front right alternating every second channel.

**FMOD_SPEAKERMAPTYPE_51_PROTOOLS**
Map a 5.1 sound to use protools L C R Ls Rs LFE mapping. Will return an error if not a 6 channel sound.

**Remarks**

For full flexibility of speaker assignments, use `Channel::setSpeakerLevels`.

**See Also**

- `FMOD_CREATESOUNDEXINFO`
- `Channel::setSpeakerLevels`
Firelight Technologies FMOD Ex
FMOD_SPEAKERMODE

These are speaker types defined for use with the System::setSpeakerMode or System::getSpeakerMode command.

Enumeration

typedef enum {
    FMOD_SPEAKERMODE_RAW,
    FMOD_SPEAKERMODE_MONO,
    FMOD_SPEAKERMODE_STEREO,
    FMOD_SPEAKERMODE_QUAD,
    FMOD_SPEAKERMODE_SURROUND,
    FMOD_SPEAKERMODE_5POINT1,
    FMOD_SPEAKERMODE_7POINT1,
    FMOD_SPEAKERMODE_SRS5_1_MATRIX,
    FMOD_SPEAKERMODE_MYEARS,
    FMOD_SPEAKERMODE_MAX
} FMOD_SPEAKERMODE;

Values

FMOD_SPEAKERMODE_RAW

There is no specific speakermode. Sound channels are mapped in order of input to output. Use System::setSoftwareFormat to specify speaker count. See remarks for more information.

FMOD_SPEAKERMODE_MONO

The speakers are monaural.

FMOD_SPEAKERMODE_STEREO

The speakers are stereo (DEFAULT).

FMOD_SPEAKERMODE_QUAD

4 speaker setup. This includes front left, front right, rear left, rear right.
**FMOD_SPEAKERMODE_SURROUND**

5 speaker setup. This includes front left, front right, center, rear left, rear right.

**FMOD_SPEAKERMODE_5POINT1**

5.1 speaker setup. This includes front left, front right, center, rear left, rear right and a subwoofer.

**FMOD_SPEAKERMODE_7POINT1**

7.1 speaker setup. This includes front left, front right, center, rear left, rear right, side left, side right and a subwoofer.

**FMOD_SPEAKERMODE_SRS5_1 MATRIX**

Stereo compatible output, embedded with surround information. SRS 5.1/Prologic/Prologic2 decoders will split the signal into a 5.1 speaker set-up or SRS virtual surround will decode into a 2-speaker/headphone setup. See remarks about limitations.

**FMOD_SPEAKERMODE_MYEARS**

Stereo output, but data is encoded using personalized HRTF algorithms. See myears.net.au

**FMOD_SPEAKERMODE_MAX**

Maximum number of speaker modes supported.

**Remarks**

These are important notes on speaker modes in regards to sounds created with FMOD SOFTWARE.

Note below the phrase 'sound channels' is used. These are the subchannels inside a sound, they are not related and have nothing to do with the FMOD class "Channel".

For example a mono sound has 1 sound channel, a stereo sound has 2 sound
channels, and an AC3 or 6 channel wav file have 6 "sound channels".

**FMOD_SPEAKERMODE_RAW**

---------------------

This mode is for output devices that are not specifically mono/stereo/quad/surround/5.1 or 7.1, but are multichannel. Use `System::setSoftwareFormat` to specify the number of speakers you want to address, otherwise it will default to 2 (stereo). Sound channels map to speakers sequentially, so a mono sound maps to output speaker 0, stereo sound maps to output speaker 0 & 1. The user assumes knowledge of the speaker order. **FMOD_SPEAKER** enumerations may not apply, so raw channel indices should be used. Multichannel sounds map input channels to output channels 1:1. Channel::setPan and Channel::setSpeakerMix do not work. Speaker levels must be manually set with `Channel::setSpeakerLevels`.

**FMOD_SPEAKERMODE_MONO**

---------------------

This mode is for a 1 speaker arrangement. Panning does not work in this speaker mode. Mono, stereo and multichannel sounds have each sound channel played on the one speaker unity. Mix behavior for multichannel sounds can be set with `Channel::setSpeakerLevels`. Channel::setSpeakerMix does not work.

**FMOD_SPEAKERMODE_STEREO**

---------------------

This mode is for 2 speaker arrangements that have a left and right speaker.

- Mono sounds default to an even distribution between left and right. They can be panned with Channel::setPan.
- Stereo sounds default to the middle, or full left in the left speaker and full right in the right speaker.
- They can be cross faded with Channel::setPan.
- Multichannel sounds have each sound channel played on each speaker at unity.
- Mix behavior for multichannel sounds can be set with `Channel::setSpeakerLevels`.
- Channel::setSpeakerMix works but only front left and right parameters are
used, the rest are ignored.

**FMOD_SPEAKERMODE_QUAD**

------------------------
This mode is for 4 speaker arrangements that have a front left, front right, rear left and a rear right speaker.
- Mono sounds default to an even distribution between front left and front right. They can be panned with Channel::setPan.
- Stereo sounds default to the left sound channel played on the front left, and the right sound channel played on the front right.
- They can be cross faded with Channel::setPan.
- Multichannel sounds default to all of their sound channels being played on each speaker in order of input.
- Mix behavior for multichannel sounds can be set with Channel::setSpeakerLevels.
- Channel::setSpeakerMix works but side left, side right, center and lfe are ignored.

**FMOD_SPEAKERMODE_SURROUND**

------------------------
This mode is for 5 speaker arrangements that have a left/right/center/rear left/rear right.
- Mono sounds default to the center speaker. They can be panned with Channel::setPan.
- Stereo sounds default to the left sound channel played on the front left, and the right sound channel played on the front right.
- They can be cross faded with Channel::setPan.
- Multichannel sounds default to all of their sound channels being played on each speaker in order of input.
- Mix behavior for multichannel sounds can be set with Channel::setSpeakerLevels.
- Channel::setSpeakerMix works but side left / side right are ignored.

**FMOD_SPEAKERMODE_5POINT1**

------------------------
This mode is for 5.1 speaker arrangements that have a left/right/center/rear left/rear right and a subwoofer speaker.
- Mono sounds default to the center speaker. They can be panned with Channel::setPan.
- Stereo sounds default to the left sound channel played on the front left, and the right sound channel played on the front right.
- They can be cross faded with Channel::setPan.
- Multichannel sounds default to all of their sound channels being played on each speaker in order of input.
- Mix behavior for multichannel sounds can be set with Channel::setSpeakerLevels.
- Channel::setSpeakerMix works but side left / side right are ignored.

**FMOD_SPEAKERMODE_7POINT1**
------------------------
This mode is for 7.1 speaker arrangements that have a left/right/center/rear left/rear right/side left/side right and a subwoofer speaker.
- Mono sounds default to the center speaker. They can be panned with Channel::setPan.
- Stereo sounds default to the left sound channel played on the front left, and the right sound channel played on the front right.
- They can be cross faded with Channel::setPan.
- Multichannel sounds default to all of their sound channels being played on each speaker in order of input.
- Mix behavior for multichannel sounds can be set with Channel::setSpeakerLevels.
- Channel::setSpeakerMix works and every parameter is used to set the balance of a sound in any speaker.

**FMOD_SPEAKERMODE_SRS5_1_MATRIX**
------------------------------------------------------
This mode is for mono, stereo, 5.1 and 6.1 speaker arrangements, as it is backwards and forwards compatible with stereo, but to get a surround effect a SRS 5.1, Prologic or Prologic 2 hardware decoder / amplifier is needed or a compatible SRS equipped device (e.g., laptop, TV, etc.) or accessory (e.g., headphone).
Pan behavior is the same as **FMOD_SPEAKERMODE_5POINT1**.

If this function is called the numoutputchannels setting in System::setSoftwareFormat is overwritten.

Output rate must be 44100, 48000 or 96000 for this to work otherwise FMOD_ERR_OUTPUT_INIT will be returned.
FMOD_SPEAKERMODE_MYEARS

This mode is for headphones. This will attempt to load a MyEars profile (see myears.net.au) and use it to generate surround sound on headphones using a personalized HRTF algorithm, for realistic 3d sound.
Pan behavior is the same as FMOD_SPEAKERMODE_7POINT1.
MyEars speaker mode will automatically be set if the speakermode is FMOD_SPEAKERMODE_STEREO and the MyEars profile exists.
If this mode is set explicitly,
FMOD_INIT_DISABLE_MYEARS_AUTODETECT has no effect.
If this mode is set explicitly and the MyEars profile does not exist,
FMOD_ERR_OUTPUT_DRIVERCALL will be returned.

See Also

- System::setSpeakerMode
- System::getSpeakerMode
- System::getDriverCaps
- System::setSoftwareFormat
- Channel::setSpeakerLevels
Firelight Technologies FMOD Ex
FMOD_SYSTEM_CALLBACKTYPE

These callback types are used with `System::setCallback`.

**Enumeration**

```c
typedef enum {
    FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED,
    FMOD_SYSTEM_CALLBACKTYPE_DEVICELOST,
    FMOD_SYSTEM_CALLBACKTYPE_MEMORYALLOCATIONFAILED,
    FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED,
    FMOD_SYSTEM_CALLBACKTYPE_BADDSPCONNECTION,
    FMOD_SYSTEM_CALLBACKTYPE_BADDSPLEVEL,
    FMOD_SYSTEM_CALLBACKTYPE_MAX
} FMOD_SYSTEM_CALLBACKTYPE;
```

**Values**

**FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED**

Called from `System::update` when the enumerated list of devices has changed.

**FMOD_SYSTEM_CALLBACKTYPEDEVICELOST**

Called from `System::update` when an output device has been lost due to control panel parameter changes and FMOD cannot automatically recover.

**FMOD_SYSTEM_CALLBACKTYPE_MEMORYALLOCATIONFAILED**

Called directly when a memory allocation fails somewhere in FMOD. (NOTE - 'system' will be NULL in this callback type.)

**FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED**

Called directly when a thread is created. (NOTE - 'system' will be NULL in this callback type.)

**FMOD_SYSTEM_CALLBACKTYPE_BADDSPCONNECTION**

Called when a bad connection was made with `DSP::addInput`. Usually called
from mixer thread because that is where the connections are made.

**FMOD_SYSTEM_CALLBACKTYPE_BADDSPLEVEL**

Called when too many effects were added exceeding the maximum tree depth of 128. This is most likely caused by accidentally adding too many DSP effects. Usually called from mixer thread because that is where the connections are made.

**FMOD_SYSTEM_CALLBACKTYPE_MAX**

Maximum number of callback types supported.

**Remarks**

Each callback has commanddata parameters passed as void* unique to the type of callback. See reference to [FMOD_SYSTEM_CALLBACK](#) to determine what they might mean for each type of callback.

**Note!** Using **FMOD_SYSTEM_CALLBACKTYPE_DEVICELISTCHANGED** (on Mac only) requires the application to be running an event loop which will allow external changes to device list to be detected by FMOD.

**Note!** The 'system' object pointer will be null for **FMOD_SYSTEM_CALLBACKTYPE_THREADCREATED** and **FMOD_SYSTEM_CALLBACKTYPE_MEMORYALLOCATIONFAILED** callbacks.

**See Also**

- [System::setCallback](#)
- [FMOD_SYSTEM_CALLBACK](#)
- [System::update](#)
- [DSP::addInput](#)
Firelight Technologies FMOD Ex
FMOD_TAGDATATYPE

List of data types that can be returned by Sound::getTag

Enumeration

typedef enum {
    FMOD_TAGDATATYPE_BINARY,
    FMOD_TAGDATATYPE_INT,
    FMOD_TAGDATATYPE_FLOAT,
    FMOD_TAGDATATYPE_STRING,
    FMOD_TAGDATATYPE_STRING_UTF16,
    FMOD_TAGDATATYPE_STRING_UTF16BE,
    FMOD_TAGDATATYPE_STRING_UTF8,
    FMOD_TAGDATATYPE_CDTOC,
    FMOD_TAGDATATYPE_MAX
} FMOD_TAGDATATYPE;

Values

FMOD_TAGDATATYPE_BINARY

FMOD_TAGDATATYPE_INT

FMOD_TAGDATATYPE_FLOAT

FMOD_TAGDATATYPE_STRING

FMOD_TAGDATATYPE_STRING_UTF16

FMOD_TAGDATATYPE_STRING_UTF16BE

FMOD_TAGDATATYPE_STRING_UTF8

FMOD_TAGDATATYPE_CDTOC

FMOD_TAGDATATYPE_MAX

Maximum number of tag datatypes supported.
See Also

- Sound::getTag
Firelight Technologies FMOD Ex
**FMOD_TAGTYPE**

List of tag types that could be stored within a sound. These include id3 tags, metadata from netstreams and vorbis/asf data.

**Enumeration**

```c
typedef enum {
    FMOD_TAGTYPE_UNKNOWN,
    FMOD_TAGTYPE_ID3V1,
    FMOD_TAGTYPE_ID3V2,
    FMOD_TAGTYPE_VORBISCOMMENT,
    FMOD_TAGTYPE_SHOUTCAST,
    FMOD_TAGTYPE_ICECAST,
    FMOD_TAGTYPEASF,
    FMOD_TAGTYPE_MIDI,
    FMOD_TAGTYPE_PLAYLIST,
    FMOD_TAGTYPE_FMOD,
    FMOD_TAGTYPE_USER,
    FMOD_TAGTYPE_MAX
} FMOD_TAGTYPE;
```

**Values**

- `FMOD_TAGTYPE_UNKNOWN`
- `FMOD_TAGTYPE_ID3V1`
- `FMOD_TAGTYPE_ID3V2`
- `FMOD_TAGTYPE_VORBISCOMMENT`
- `FMOD_TAGTYPE_SHOUTCAST`
- `FMOD_TAGTYPE_ICECAST`
- `FMOD_TAGTYPEASF`
- `FMOD_TAGTYPE_MIDI`
- `FMOD_TAGTYPE_PLAYLIST`
**FMOD_TAGTYPE_FMOD**

**FMOD_TAGTYPE_USER**

**FMOD_TAGTYPE_MAX**

Maximum number of tag types supported.

**See Also**

- [Sound::getTag](#)
Firelight Technologies FMOD Ex
FMOD_THREAD

Values for the thread members of the FMOD_360_EXTRADRIVERDATA structure.

Enumeration

typedef enum {
    FMOD_THREAD_DEFAULT,
    FMOD_THREAD_CORE0THREAD0,
    FMOD_THREAD_CORE0THREAD1,
    FMOD_THREAD_CORE1THREAD0,
    FMOD_THREAD_CORE1THREAD1,
    FMOD_THREAD_CORE2THREAD0,
    FMOD_THREAD_CORE2THREAD1,
    FMOD_THREAD_MAX
} FMOD_THREAD;

Values

FMOD_THREAD_DEFAULT

Use FMOD recommended default thread assignment.

FMOD_THREAD_CORE0THREAD0

Thread will be created on HW Thread 0 of Core 0

FMOD_THREAD_CORE0THREAD1

Thread will be created on HW Thread 1 of Core 0

FMOD_THREAD_CORE1THREAD0

Thread will be created on HW Thread 0 of Core 1

FMOD_THREAD_CORE1THREAD1

Thread will be created on HW Thread 1 of Core 1
**FMOD_THREAD_CORE2THREAD0**

Thread will be created on HW Thread 0 of Core 2. Default for all threads (except recording)

**FMOD_THREAD_CORE2THREAD1**

Thread will be created on HW Thread 1 of Core 2. Default for recording thread

**FMOD_THREAD_MAX**

Maximum number of thread options supported.

**See Also**

- [FMOD_360_EXTRADRIVERDATA](#)
Firelight Technologies FMOD Ex
C++ interfaces

EventSystem  EventProject  
EventGroup  
Event  
EventParameter  
EventCategory  
EventReverb  
EventQueue  
EventQueueEntry  
MusicSystem  
MusicPrompt
Firelight Technologies FMOD Ex
EventSystem Interface

EventSystem::createEventQueue  EventSystem::createEventQueueEntry
EventSystem::createReverb
EventSystem::get3DListenerAttributes
EventSystem::get3DNumListeners
EventSystem::getCategory
EventSystem::getCategoryByIndex
EventSystem::getEvent
EventSystem::getEventByGUID
EventSystem::getEventByGUIDString
EventSystem::getEventBySystemID
EventSystem::getGroup
EventSystem::getInfo
EventSystem::getLanguage
EventSystem::getMemoryInfo
EventSystem::getMusicCategory
EventSystem::getMusicSystem
EventSystem::getNumCategories
EventSystem::getNumEvents
EventSystem::getNumProjects
EventSystem::getNumReverbPresets
EventSystem::getProject
EventSystem::getProjectByIndex
EventSystem::getReverbAmbientProperties
EventSystem::getReverbPreset
EventSystem::getReverbPresetByIndex
EventSystem::getReverbProperties
EventSystem::getSystemObject
EventSystem::getUserData
EventSystem::getVersion
EventSystem::init
EventSystem::load
EventSystem::preloadFSB
EventSystem::registerDSP
EventSystem::release
EventSystem::set3DListenerAttributes
EventSystem::set3DNumListeners
EventSystem::setLanguage
EventSystem::setMediaPath
EventSystem::setPluginPath
EventSystem::setReverbAmbientProperties
EventSystem::setReverbProperties
EventSystem::setUserData
EventSystem::unload
EventSystem::unloadFSB
EventSystem::update
Firelight Technologies FMOD Ex
EventSystem::createEventQueue

Create a new EventQueue object. EventQueue objects allow you to queue up multiple events and let FMOD play them back for you. They make it easy to do things like dynamic dialog commentary, for instance.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::createEventQueue(
    EventQueue ** queue
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_CreateEventQueue(
    FMOD_EVENTSYSTEM * eventSystem,
    FMOD_EVENTQUEUE ** queue
);
```

Parameters

`queue`

Address of a pointer that will receive a pointer to the newly created EventQueue object

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [EventSystem::createEventQueueEntry](#)
- EventQueue::release
Firelight Technologies FMOD Ex
EventSystem::createEventQueueEntry

Create a new EventQueueEntry object for the specified Event.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::createEventQueueEntry(
    Event * event,
    EventQueueEntry ** entry
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_CreateEventQueueEntry(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_EVENT * event,
    FMOD_EVENTQUEUEENTRY ** entry
);
```

Parameters

`event`

The Event to create an EventQueueEntry for. NOTE: This Event must have been retrieved with the FMOD_EVENT_INFOONLY flag set

`entry`

Address of a pointer that will receive a pointer to the newly created EventQueueEntry object

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Remarks

The event that gets passed to this function must have been retrieved with the \texttt{FMOD\_EVENT\_INFOONLY} flag. When an event queue entry reaches the head of the queue, FMOD will get a real event instance using the \texttt{FMOD\_EVENT\_INFOONLY} handle.

\textbf{NOTE:} The event that gets passed to this function must be set up so that it ends naturally (i.e. without further programmer interaction) when played. Generally, this means that the event should be a "oneshot" event. If the event doesn't end naturally then the queue will get blocked.

See Also

- \texttt{EventSystem::createEventQueue}
- \texttt{EventQueue::add}
- \texttt{EventQueueEntry::release}
Firelight Technologies FMOD Ex
### EventSystem::createReverb

Creates a 'virtual reverb' object. This object reacts to 3d location and morphs the reverb environment based on how close it is to the reverb object's center. Multiple reverb objects can be created to achieve a multi-reverb environment.

#### C++ Syntax

```cpp
FMOD_RESULT EventSystem::createReverb(
    EventReverb ** reverb
);
```

#### C Syntax

```c
FMOD_RESULT FMOD_EventSystem_CreateReverb(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_EVENTREVERB ** reverb
);
```

#### Parameters

- `reverb`

#### Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

#### Remarks

The 3D reverb object is a sphere having 3D attributes (position, minimum distance, maximum distance) and reverb properties. The properties and 3D attributes of all reverb objects collectively determine, along with the listener's position, the settings of and input gains into a single 3D
reverb DSP.
Please note that this only applies to software channels. When the listener is within the sphere of effect of one or more 3d reverbs, the listener's 3D reverb properties are a weighted combination of such 3d reverbs. When the listener is outside all of the reverbs, the 3D reverb setting is set to the default ambient reverb setting.

Use **EventSystem::setReverbAmbientProperties** to set a 'background' default reverb environment. This is a reverb that will be morphed to if the listener is not within any virtual reverb zones.
By default the ambient reverb is set to 'off'.
Creating multiple reverb objects does not impact performance. These are 'virtual reverbs'. There will still be only 1 physical reverb DSP running that just morphs between the different virtual reverbs. **EventSystem::setReverbProperties** can still be used in conjunction with the 3d based virtual reverb system. This allows 2d sounds to have reverb. If this call is used at the same time virtual reverb objects are active, 2 physical reverb dsps will be used, incurring a small memory and cpu hit.

**See Also**

- **EventSystem::setReverbAmbientProperties**
- **EventSystem::getReverbAmbientProperties**
- **EventSystem::setReverbProperties**
- **EventSystem::getReverbProperties**
- **EventReverb::release**
Firelight Technologies FMOD Ex
EventSystem::get3DListenerAttribute

This retrieves the position, velocity and orientation of the specified 3D sound listener.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::get3DListenerAttributes(
    int listener,
    FMOD_VECTOR * pos,
    FMOD_VECTOR * vel,
    FMOD_VECTOR * forward,
    FMOD_VECTOR * up
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_Get3DListenerAttributes(
    FMOD_EVENTSYSTEM * eventsystem,
    int listener,
    FMOD_VECTOR * pos,
    FMOD_VECTOR * vel,
    FMOD_VECTOR * forward,
    FMOD_VECTOR * up
);
```

Parameters

\textit{listener}

Listener ID in a multi-listener environment. Specify 0 if there is only 1 listener.

\textit{pos}

Address of a variable that receives the position of the listener in world space, measured in distance units. Optional. Specify 0 to ignore.

\textit{vel}

Address of a variable that receives the velocity of the listener measured in distance units \textbf{per second}. Optional. Specify 0 to ignore.
**forward**

Address of a variable that receives the forwards orientation of the listener. Optional. Specify 0 to ignore.

**up**

Address of a variable that receives the upwards orientation of the listener. Optional. Specify 0 to ignore.

### Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

### See Also

- [EventSystem::set3DListenerAttributes](#)
- [FMOD_VECTOR](#)
Firelight Technologies FMOD Ex
EventSystem::get3DNumListeners

Retrieves the number of 3D listeners.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::get3DNumListeners(
    int * numlisteners
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_Get3DNumListeners(
    FMOD_EVENTSYSTEM * eventsystem,
    int * numlisteners
);
```

Parameters

`numlisteners`

Address of a variable that receives the current number of 3D listeners in the 3D scene.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventSystem::set3DNumListeners`
Firelight Technologies FMOD Ex
EventSystem::getCategory

Retrieve an event category object by name.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getCategory(
    const char * name,
    EventCategory ** category
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetCategory(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * name,
    FMOD_EVENTCATEGORY ** category
);
```

Parameters

`name`

The name of an event category within this event system. Specify "master" to retrieve the master event category.

`category`

Address of a variable to receive the selected event category within this event system.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Sub-categories can be retrieved by specifying their full path relative to the master category e.g. "vehicles/cars/racers".
Note: Don't specify "master/vehicles/cars/racers", just use "vehicles/cars/racers".

See Also

- EventSystem::getCategoryByIndex
- EventSystem::getMusicCategory
- EventSystem::getNumCategories
Firelight Technologies FMOD Ex
EventSystem::getCategoryByIndex

Retrieve an event category object by index.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getCategoryByIndex(
    int index,
    EventCategory ** category
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetCategoryByIndex(
    FMOD_EVENTSYSTEM * eventsystem,
    int index,
    FMOD_EVENTCATEGORY ** category
);
```

Parameters

`index`

The index of an event category within this event system object. Indices are 0 based. Specify -1 to retrieve the master event category.

`category`

Address of a variable to receive the selected event category within this event system.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- EventSystem::getCategory
- EventSystem::getMusicCategory
- EventSystem::getNumCategories
Firelight Technologies FMOD Ex
EventSystem::getEvent

Retrieve an event object by name.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getEvent(
    const char * name,
    FMOD_EVENT_MODE mode,
    Event ** event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetEvent(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * name,
    FMOD_EVENT_MODE mode,
    FMOD_EVENT ** event
);
```

Parameters

name

The name of an event within this event system. Note: name must include full path including project name and any event group names e.g. "myproject/group1/group2/myevent"

mode

If not already loaded with EventGroup::loadEventData, this `FMOD_EVENT_MODE` flag will determine if data should be loaded from disk synchronously or asynchronously. It also allows an 'info only' event retrieval (events that can't be played) and can be used to produce an error on disk access.

event

Address of a variable to receive the selected event within this event system.
**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

What is an event?  
An event is the leaf of the event group tree. It is the actual sound to be played with complex behaviour designed by the sound designer.
In this case we are retrieving an event from an event group, so with the "filters" group we could get the echo event with "echo" as the name parameter.

If the programmer does not know which events are available, the sound designer tool can output a programmer report that lists the event group's events with the appropriate names and indices listed alongside them.

Note!
- An event is retrieved from a pool of events (created earlier if FMOD_EVENT_CACHEEVENTS flag was set in EventSystem::getGroup / EventGroup::getGroup).
- Data may not be loaded from the disk for this event, so this event may trigger disk access. If you wish to pre-emp this use EventGroup::loadEventData first. -
The pool of events has a size determined by the 'max playbacks' property in the FMOD Designer tool in the event's property sheet.
- The pointer to will be getting will be a pointer to one of these event instances.
- If you call this function more times than there are event instances, then an invent handle may be stolen, or may fail. This behaviour also determined by the sound designer. The behaviour may be to steal the oldest event in the pool, steal the quietest event in the pool, or simply fail this getEvent and return null as the event handle.

See Also

- [EventSystem::getGroup](#)
- [FMOD_EVENT_MODE](#)
Firelight Technologies FMOD Ex
EventSystem::getEventByGUID

Retrieve an event object by GUID.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getEventByGUID(
    const FMOD_GUID * guid,
    FMOD_EVENT_MODE mode,
    Event ** event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetEventByGUID(
    FMOD_EVENTSYSTEM * eventsystem,
    const FMOD_GUID * guid,
    FMOD_EVENT_MODE mode,
    FMOD_EVENT ** event
);
```

Parameters

`guid`

The GUID of the desired event.

`mode`

If not already loaded with EventGroup::loadEventData, this `FMOD_EVENT_MODE` flag will determine if data should be loaded from disk synchronously or asynchronously. It also allows an 'info only' event retrieval (events that can't be played) and can be used to produce an error on disk access.

`event`

Address of a variable to receive a handle for the desired event.
Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

The GUID for an event can be found in the programmer report, in the project header file or in the event properties in FMOD Designer. It can also be retrieved using **Event::getInfo** by setting the guid member of the **FMOD_EVENT_INFO** structure.

**Note:** This function will not work unless **FMOD_EVENT_INIT_USEGUIDS** is passed to **EventSystem::init**.

See Also

- **EventSystem::getEventByGUIDString**
- **EventSystem::getEventBySystemID**
- **EventGroup::loadEventData**
- **Event::getInfo**
- **EventSystem::init**
- **FMOD_EVENT_INFO**
- **FMOD_EVENT_MODE**
- **FMOD_GUID**
Firelight Technologies FMOD Ex
EventSystem::getEventByGUIDString

Retrieve an event object by GUID.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getEventByGUIDString(
    const char * guid,
    FMOD_EVENT_MODE mode,
    Event ** event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetEventByGUIDString(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * guid,
    FMOD_EVENT_MODE mode,
    FMOD_EVENT ** event
);
```

Parameters

**guid**

The GUID of the desired event. See remarks for the format.

**mode**

If not already loaded with `EventGroup::loadEventData`, this `FMOD_EVENT_MODE` flag will determine if data should be loaded from disk synchronously or asynchronously. It also allows an 'info only' event retrieval (events that can't be played) and can be used to produce an error on disk access.

**event**

Address of a variable to receive a handle for the desired event.
Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

The GUID for an event can be found in the programmer report, in the project header file or in the event properties in FMOD Designer. It can also be retrieved using `Event::getInfo` by setting the guid member of the **FMOD_EVENT_INFO** structure.

**Note:** This function will not work unless **FMOD_EVENT_INIT_USE_GUIDS** is passed to `EventSystem::init`.

This function expects a GUID string formatted as "{11111111-2222-3333-4444-555555555555}", where the digits indicate the field number.

<table>
<thead>
<tr>
<th>Field</th>
<th>Hexadecimal Digits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>The Data1 value of the GUID</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>The Data2 value of the GUID</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>The Data3 value of the GUID</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>The initial two bytes of the Data4 value of the GUID</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>The remaining six bytes of the Data4 value of the GUID</td>
</tr>
</tbody>
</table>

See Also

- `EventSystem::getEventByGUID`
- `EventSystem::getEventBySystemID`
- `EventGroup::loadEventData`
- `Event::getInfo`
- `EventSystem::init`
- **FMOD_EVENT_INFO**
- **FMOD_EVENT_MODE**
EventSystem::getEventBySystemID

Retrieve an event object by system wide unique identifier. All loaded events can be enumerated with this function and EventSystem::getNumEvents.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getEventBySystemID(
    unsigned int systemid,
    FMOD_EVENT_MODE mode,
    Event ** event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetEventBySystemID(
    FMOD_EVENTSYSTEM * eventsystem,
    unsigned int systemid,
    FMOD_EVENT_MODE mode,
    FMOD_EVENT ** event
);
```

Parameters

**systemid**

**mode**

If not already loaded with EventGroup::loadEventData, this **FMOD_EVENT_MODE** flag will determine if data should be loaded from disk synchronously or asynchronously. It also allows an 'info only' event retrieval (events that can't be played) and can be used to produce an error on disk access.

**event**

Address of a variable to receive the selected event within this event system.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

**What is an event?**
An event is the leaf of the event group tree. It is the actual sound to be played with complex behaviour designed by the sound designer.

In this case we are retrieving an event from an **event group**, so with the "**filters**" group we could get the echo event with "**echo**" as the name parameter.
If the programmer does not know which events are available, the sound designer tool can output a programmer report that lists the event group's events with the appropriate names and indices listed alongside them.

**Note!**
- An event is retrieved from a pool of events (created earlier if FMOD_EVENT_CACHEEVENTS flag was set in EventSystem::getGroup / EventGroup::getGroup).
- Data may not be loaded from the disk for this event, so this event may trigger disk access. If you wish to pre-emp this use EventGroup::loadEventData first.
- The pool of events has a size determined by the 'max playbacks' property in the FMOD Designer tool in the event's property sheet.
- The pointer to will be getting will be a pointer to one of these event instances.
- If you call this function more times than there are event instances, then an invent handle may be stolen, or may fail. This behaviour also determined by the sound designer. The behaviour may be to steal the oldest event in the pool, steal the quietest event in the pool, or simply fail this getEvent and return null as the event handle.

**See Also**

- [EventSystem::getNumEvents](#)
- [FMOD_EVENT_MODE](#)
Firelight Technologies FMOD Ex
EventSystem::getGroup

Retrieves an event group object by name.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getGroup(
    const char * name,
    bool cacheevents,
    EventGroup ** group
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetGroup(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * name,
    FMOD_BOOL cacheevents,
    FMOD_EVENTGROUP ** group
);
```

Parameters

**name**

The name of an event group that belongs to this event system. Note: name must include full path including project name and any event group names e.g. "myproject/group1/group2"

**cacheevents**

If cacheevents is true then all event instances within this event group will be pre-allocated so that there are no memory allocs when getEvent is called.

**group**

Address of a variable to receive the selected event group within this event system.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

What is an event group?
An event group is a "folder" that stores events or sub-folders. With these folders a hierarchical tree can be built to store events in a more logical manner.

In this case we are retrieving an event group from another `event group`, so if this
event group object was "examples" we could then get the event group "filters" with "filters" as the name parameter. In this example "filters" is the only sub-group below "examples" so no other sub-groups are available here.

If the programmer does not know which sub-groups are available or which sub-group index matches which sub-group name, the sound designer tool can output a programmer report that lists the group's sub-groups with the appropriate names and indices listed alongside them. The only benefit of retrieving an object by index is that it is slightly faster to do so than to retrieve it by name.

See Also

- EventGroup::getGroup
- EventGroup::getGroupByIndex
Firelight Technologies FMOD Ex
EventSystem::getInfo

Retrieves information about the event system.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getInfo(
    FMOD_EVENT_SYSTEMINFO * info
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetInfo(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_EVENT_SYSTEMINFO * info
);
```

Parameters

`info`

Address of an `FMOD_EVENT_SYSTEMINFO` structure to receive event system information.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `FMOD_EVENT_SYSTEMINFO`
Firelight Technologies FMOD Ex
EventSystem::getLanguage

Retrieves the current language set by the user.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getLanguage(
    char * language
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetLanguage(
    FMOD_EVENTSYSTEM * eventsystem,
    char * language
);
```

Parameters

*language*

Address of a variable to receive the currently set language inside the event system.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Note that the maximum string length that FMOD will write into is 256 bytes. If you're not sure of the language string length, give this function a pointer to a buffer of at least 256 bytes.
See Also

- EventSystem::setLanguage
Firelight Technologies FMOD Ex
EventSystem::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetMemoryInfo(
    FMOD_EVENTSYSTEM * eventsystem,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

event_memorybits

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

memoryused

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated
FMOD_MEMORY_USAGE_DETAILS structure to be filled with detailed
memory usage information about this object.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the
FMOD_RESULT enumeration.

**Remarks**

See [System::getMemoryInfo](#) for more details.

**See Also**

- [FMOD_MEMBITS](#)
- [FMOD_EVENT_MEMBITS](#)
- [FMOD_MEMORY_USAGE_DETAILS](#)
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
EventSystem::getMusicCategory

Retrieve the music event category.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getMusicCategory(
    EventCategory ** category
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetMusicCategory(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_EVENTCATEGORY ** category
);
```

Parameters

`category`

Address of a variable to receive the music event category within this event system.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventSystem::getCategory`
- `EventSystem::getCategoryByIndex`
- `EventSystem::getNumCategories`
Firelight Technologies FMOD Ex
EventSystem::getMusicSystem

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getMusicSystem(
    MusicSystem ** musicsystem
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetMusicSystem(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_MUSICSYSTEM ** musicsystem
);
```

Parameters

`musicsystem`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Firelight Technologies FMOD Ex
EventSystem::getNumCategories

Retrieve the number of categories for the event system.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getNumCategories(
    int * numcategories
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetNumCategories(
    FMOD_EVENTSYSTEM * eventsystem,
    int * numcategories
);
```

Parameters

`numcategories`

Address of a variable to receive the number of categories for this event system.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventSystem::getCategoryByIndex`
- `EventSystem::getCategory`
- `EventSystem::getMusicCategory`
**EventSystem::getNumEvents**

Gets the total number of unique events loaded into the system at once, including those from different EventProjects. Mainly used for enumeration of all events loaded into the system. This can be done in conjunction with **EventSystem::getEventBySystemID**.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::getNumEvents(
    int * numevents
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_GetNumEvents(
    FMOD_EVENTSYSTEM * eventsystem,
    int * numevents
);
```

**Parameters**

`numevents`

Pointer to a integer to retrieve the current number of unique events.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- **EventSystem::getEventBySystemID**
Firelight Technologies FMOD Ex
EventSystem::getNumProjects

Retrieve the number of event projects within the top level event system.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getNumProjects(
    int * numprojects
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetNumProjects(
    FMOD_EVENTSYSTEM * eventsystem,
    int * numprojects
);
```

Parameters

`numprojects`

Address of a variable to receive the number of event projects within the top level event system.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventSystem::getProjectByIndex
Firelight Technologies FMOD Ex
EventSystem::getNumReverbPresets

Retrieve the number of reverb presets defined by the sound designer.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getNumReverbPresets(
    int * numpresets
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetNumReverbPresets(
    FMOD_EVENTSYSTEM * eventsystem,
    int * numpresets
);
```

Parameters

`numpresets`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Use this in conjunction with `EventSystem::getReverbPresetByIndex` to enumerate all sound designer specified reverb presets.

See Also
• `EventSystem::getReverbPresetByIndex`
• `EventSystem::getReverbPreset`
Firelight Technologies FMOD Ex
EventSystem::getProject

Retrieve an event project object by name.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getProject(
    const char * name,
    EventProject ** project
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetProject(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * name,
    FMOD_EVENTPROJECT ** project
);
```

Parameters

name

The name of an event project within this event system.

project

Address of a variable to receive the selected event project within this event system.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
See Also

- EventSystem::getProjectByIndex
- EventSystem::load
Firelight Technologies FMOD Ex
EventSystem::getProjectByIndex

Retrieve an event project object by index.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::getProjectByIndex(
    int index,
    EventProject ** project
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_GetProjectByIndex(
    FMOD_EVENTSYSTEM * eventsystem,
    int index,
    FMOD_EVENTPROJECT ** project
);
```

**Parameters**

*index*

The index of an event project within this event system object. Indices are 0 based.

*project*

Address of a variable to receive the selected event project within this event system.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
See Also

- EventSystem::getProject
- EventSystem::getNumProjects
- EventSystem::load
Firelight Technologies FMOD Ex
EventSystem::getReverbAmbientProperties

Retrieves the default reverb environment for the virtual reverb system.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getReverbAmbientProperties(
    FMOD_REVERB_PROPERTIES * props
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetReverbAmbientProperties(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_REVERB_PROPERTIES * props
);
```

Parameters

`props`

Return Values

If the function succeeds then the return value is `FMOD_OK`.  If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

By default the ambient reverb is set to 'off'. This is the same as `FMOD_REVERB_PRESET_OFF`.

See Also
- **FMOD_REVERB_PROPERTIES**
- **EventSystem::setReverbAmbientProperties**
- **EventSystem::createReverb**
Firelight Technologies FMOD Ex
EventSystem::getReverbPreset

Retrieves a reverb property structure containing a reverb preset created by the sound designer, by name.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getReverbPreset(
    const char* name,
    FMOD_REVERB_PROPERTIES* props,
    int* index
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetReverbPreset(
    FMOD_EVENTSYSTEM* eventsystem,
    const char* name,
    FMOD_REVERB_PROPERTIES* props,
    int* index
);
```

Parameters

**name**

The name of an event reverb within this event system.

**props**

**index**

Address of a variable to receive the index of the preset in the reverb list. Optional. This index is the index used in `EventSystem::getReverbPresetByIndex`. Specify 0 or NULL to ignore.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

Use the retrieved **FMOD_REVERB_PROPERTIES** structure to pass to `EventSystem::setReverbProperties`, `EventSystem::setReverbAmbientProperties` or `EventReverb::setProperties`. Reverb presets can also be retrieved with `EventSystem::getReverbPresetByIndex`.

**See Also**

- **FMOD_REVERB_PROPERTIES**
- `EventSystem::getReverbPresetByIndex`
- `EventSystem::setReverbProperties`
- `EventSystem::setReverbAmbientProperties`
- `EventReverb::setProperties`
- `EventReverb::release`
Firelight Technologies FMOD Ex
EventSystem::getReverbPresetByIndex

Retrieves a reverb property structure containing a reverb preset created by the sound designer, by index instead of name.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getReverbPresetByIndex(
    const int index,
    FMOD_REVERB_PROPERTIES * props,
    char ** name
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetReverbPresetByIndex(
    FMOD_EVENTSYSTEM * eventsystem,
    const int index,
    FMOD_REVERB_PROPERTIES * props,
    char ** name
);
```

Parameters

`index`

The index of an event reverb within this event system object. Indices are 0 based.

`props`

`name`

Address of a variable to receive a pointer to the name of the preset. Optional. This index is the name used in `EventSystem::getReverbPreset`. Specify 0 or NULL to ignore.

Return Values
If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration. Use the retrieved `FMOD_REVERB_PROPERTIES` structure to pass to `EventSystem::setReverbProperties`, `EventSystem::setReverbAmbientProperties` or `EventReverb::setProperties`.

**Remarks**

All reverbs can be enumerated by using this function in conjunction with `EventSystem::getNumReverbPresets`. Reverb presets can also be retrieved with `EventSystem::getReverbPreset`.

**See Also**

- `FMOD_REVERB_PROPERTIES`
- `EventSystem::getReverbPreset`
- `EventSystem::getNumReverbPresets`
- `EventReverb::release`
Firelight Technologies FMOD Ex
EventSystem::getReverbProperties

Retrieves the current reverb environment.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getReverbProperties(
    FMOD_REVERB_PROPERTIES * props
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetReverbProperties(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_REVERB_PROPERTIES * props
);
```

Parameters

`props`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventSystem::setReverbProperties
- Event::setReverbProperties
- Event::getReverbProperties
Firelight Technologies FMOD Ex
EventSystem::getSystemObject

Retrieve the event system's internal FMOD::System object for the low level FMOD Ex API.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getSystemObject(
    FMOD::System ** system
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetSystemObject(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_SYSTEM ** system
);
```

Parameters

`system`

Address of a pointer to receive the FMOD::System pointer.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Note! This should generally not be used unless you are trying to add features that the sound designer cannot provide!
The aim of this API is to give the sound designer the control over the sound behaviour. If there are things missing from the EventSystem API that could be included contact FMOD support at support@fmod.org and it will be considered for addition.
Firelight Technologies FMOD Ex
EventSystem::getUserData

Retrieves the user value that that was set by calling the EventSystem::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetUserData(
    FMOD_EVENTSYSTEM * eventsystem,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the EventSystem::setUserData function.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventSystem::setUserData
Firelight Technologies FMOD Ex
EventSystem::getVersion

Returns the current version of the event system being used.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::getVersion(
    unsigned int * version
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_GetVersion(
    FMOD_EVENTSYSTEM * eventsystem,
    unsigned int * version
);
```

Parameters

version

Address of a variable that receives the current event system version.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

The version is a 32bit hexadecimal value formatted as 16:8:8, with the upper 16bits being the major version, the middle 8bits being the minor version and the bottom 8bits being the development version. For example a value of 00010106h is equal to 1.01.06.
See Also

- EventSystem::init
Firelight Technologies FMOD Ex
EventSystem::init

Initializes the event system object, FMOD system object and the sound device. This has to be called at the start of the user's program.

You must create an event system object with EventSystem_Create.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::init(
    int maxchannels,
    FMOD_INITFLAGS flags,
    void * extradriverdata,
    FMOD_EVENT_INITFLAGS eventflags
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_Init(
    FMOD_EVENTSYSTEM * eventsystem,
    int maxchannels,
    FMOD_INITFLAGS flags,
    void * extradriverdata,
    FMOD_EVENT_INITFLAGS eventflags
);
```

Parameters

`maxchannels`

The maximum number of channels to be used in FMOD. They are also called 'virtual channels' as you can play as many of these as you want, even if you only have a small number of hardware or software voices. See remarks for more.

`flags`

See `FMOD_INITFLAGS`. This can be a selection of flags bitwise OR'ed together to change the behaviour of FMOD at initialization time.

`extradriverdata`
Driver specific data that can be passed to the output plugin. For example the filename for the wav writer plugin. See `FMOD_OUTPUTTYPE` for what each output mode might take here. Optional. Specify 0 to ignore.

*eventflags*

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See FMOD Ex documentation for details on `FMOD_INITFLAGS` etc.

**See Also**

- `EventSystem_Create`
Firelight Technologies FMOD Ex
EventSystem::load

Loads an event file (.fev).

C++ Syntax

```cpp
FMOD_RESULT EventSystem::load(
    const char * name_or_data,
    FMOD_EVENT_LOADINFO * loadinfo,
    EventProject ** project
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_Load(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * name_or_data,
    FMOD_EVENT_LOADINFO * loadinfo,
    FMOD_EVENTPROJECT ** project
);
```

Parameters

name_or_data

Filename of the event file to be loaded or pointer to memory block if FMOD_EVENT_LOADINFO is provided and has a non-zero "loadfrommemory_length" field.

loadinfo

Pointer to an FMOD_EVENT_LOADINFO structure which lets the user provide extended information about loading the file. Optional.Specify 0 or NULL to ignore.

project

Address of a variable to receive the specified project object. Optional. Specify 0 or NULL to ignore.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

Loading the event data file will not open any FSB files or allocate any memory for events. This is done with **EventGroup::loadEventData** (to load FSB data) and **EventProject::getGroup** / **EventProject::getGroupByIndex** / **EventGroup::getGroup** / **EventGroup::getGroupByIndex** to allocate memory for the event instances so that they can be played. To load a .fev file from memory, pass a pointer to the memory block in "name_or_data" and provide an FMOD_EVENT_LOADINFO structure with the "loadfrommemory_length" field set to the length of the memory block. The memory block can be freed immediately after this function returns.

Note: Currently FMOD can only load one project that contains music data at a time. Music data is created whenever you do work in the music tab in FMOD Designer. **FMOD_ERR_UNSUPPORTED** will be returned if you try to load more than one project containing music data.

See Also

- **EventSystem::setMediaPath**
- **EventGroup::loadEventData**
- **EventSystem::getProject**
- **EventSystem::getProjectByIndex**
- **EventSystem::unload**
- **EventGroup::getGroup**
- **EventGroup::getGroupByIndex**
- **EventProject::release**
**EventSystem::preloadFSB**

For users that want to pre-load static sample FSB files into memory, this function can be used to stop FMOD from loading any FSB with the same filename from disk.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::preloadFSB(
    const char * filename,
    int streaminstance,
    FMOD::Sound * sound,
    bool unloadprevious
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_PreloadFSB(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * filename,
    int streaminstance,
    FMOD_SOUND * sound,
    FMOD_BOOL unloadprevious
);
```

**Parameters**

**filename**

File name that FMOD event system would use to load. FMOD will compare this string against media path + fsb filename or just fsb filename by itself.

**streaminstance**

If a bank is streaming, this is the instance number out of the total streams instances that might be possibly used for this bank (see 'Max Streams' in the wave banks settings). For sample based banks just supply 0. See remarks for more.

**sound**
Pointer to a preloaded FSB file, with System::createSound.

unloadprevious

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

For data that has already been loaded into memory by the user, use FMOD_OPENMEMORY_POINT when creating a sound to stop it duplicating memory.

If you use FMOD_OPENMEMORY_POINT, do not free the memory until you have called EventSystem::unloadFSB.
On Playstation 2, you cannot 'point' to VAG data in EE ram. You will have to use FMOD_OPENMEMORY instead, then you can free the data from EE ram.

More on 'streaminstance'. If an event refers to a bank more than once, for example it has 'Max Playbacks' set to something higher than 1, say 4, then you will need to open the bank 4 times and call this function 4 times, each with a different instance number to stop FMOD from trying to load it from disk. This number of instances will usually correspond to the number of streams specified in the wavebank screen under 'Max streams'.
Note though that the value set in designer might be too high. It is worth being careful and counting what events use the bank at the same time otherwise you might be preloading too many streams that are not used, wasting memory.
As an example if you only called this function once, the first instance of the stream would be preloaded, but the other 3 would attempt to load from disk as if it was not preloaded.
streaminstance has no relevance to sample based banks, they are only ever opened once inside fmod, so 0 can be used here and only preload it once.
If you have 1 event with 1 sound that refers to the bank once, just supply 0.
See Also

- EventSystem::unloadFSB
- System::createSound
- System::createStream
- FMOD_MODE
Firelight Technologies FMOD Ex
EventSystem::registerDSP

Register a user-defined DSP effect for use with the EventSystem. This function allows you to register statically-linked DSP effects. Once registered, you can create instances of the DSP effect by using System::createDSPByPlugin.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::registerDSP(
    FMOD_DSP_DESCRIPTION * description,
    unsigned int * handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_RegisterDSP(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_DSP_DESCRIPTION * description,
    unsigned int * handle
);
```

Parameters

* description
  
  Address of an FMOD_DSP_DESCRIPTION structure, containing information about the DSP effect.

* handle
  
  Address of a variable to receive the plugin handle of the newly-registered DSP effect.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
See Also

- **FMOD_DSP_DESCRIPTION**
- **System::registerDSP**
- **System::createDSPByPlugin**
- **System::getNumPlugins**
- **System::getPluginInfo**
Firelight Technologies FMOD Ex
**EventSystem::release**

Closes and frees an event system object.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::release();
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_Release(FMOD_EVENTSYSTEM * eventsystem);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This will free the event system object and everything created under it.

**See Also**

- `EventSystem_Create`
- `EventSystem::init`
Firelight Technologies FMOD Ex
EventSystem::set3DListenerAttributes

This updates the position, velocity and orientation of the specified 3D sound listener.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::set3DListenerAttributes(
    int listener,
    const FMOD_VECTOR * pos,
    const FMOD_VECTOR * vel,
    const FMOD_VECTOR * forward,
    const FMOD_VECTOR * up
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_Set3DListenerAttributes(
    FMOD_EVENTSYSTEM * eventsystem,
    int listener,
    const FMOD_VECTOR * pos,
    const FMOD_VECTOR * vel,
    const FMOD_VECTOR * forward,
    const FMOD_VECTOR * up
);
```

Parameters

**listener**

Listener ID in a multi-listener environment. Specify 0 if there is only 1 listener.

**pos**

Address of a variable that receives the position of the listener in world space, measured in distance units. You can specify 0 or NULL to not update the position.

**vel**

Address of a variable that receives the velocity of the listener measured in
distance units **per second**. You can specify 0 or NULL to not update the velocity of the listener.

**forward**

Address of a variable that receives the forwards orientation of the listener. This vector must be of unit length and perpendicular to the up vector. You can specify 0 or NULL to not update the forwards orientation of the listener.

**up**

Address of a variable that receives the upwards orientation of the listener. This vector must be of unit length and perpendicular to the forwards vector. You can specify 0 or NULL to not update the upwards orientation of the listener.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

By default, FMOD uses a left-handed co-ordinate system. This means +X is right, +Y is up, and +Z is forwards.

To change this to a right-handed coordinate system, use **FMOD_INIT_3D_RIGHTHANDED**. This means +X is right, +Y is up, and +Z is backwards or towards you.

To map to another coordinate system, flip/negate and exchange these values.

Orientation vectors are expected to be of UNIT length. This means the magnitude of the vector should be 1.0.

A 'distance unit' is specified by the sound designer in the FMOD Designer tool. By default this is set to meters which is a distance scale of 1.0.
Always remember to use **units per second**, not units per frame as this is a common mistake and will make the doppler effect sound wrong. For example, Do not just use (pos - lastpos) from the last frame's data for velocity, as this is not correct. You need to time compensate it so it is given in units per **second**.
You could alter your pos - lastpos calculation to something like this.

```
vel = (pos-lastpos) / time_taken_since_last_frame_in_seconds.
```

I.e. at 60fps the formula would look like this `vel = (pos-lastpos) / 0.0166667`.

**See Also**

- EventSystem::get3DListenerAttributes
- FMOD_INITFLAGS
- FMOD_VECTOR
Firelight Technologies FMOD Ex
**EventSystem::set3DNumListeners**

Sets the number of 3D 'listeners' in the 3D sound scene. This function is useful mainly for split-screen game purposes.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::set3DNumListeners(
    int numlisteners
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_Set3DNumListeners(
    FMOD_EVENTSYSTEM * eventsystem,
    int numlisteners
);
```

**Parameters**

`numlisteners`

Number of listeners in the scene. Valid values are from 1-4 inclusive. Default = 1.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

If the number of listeners is set to more than 1, then panning and doppler are turned off. All sound effects will be mono.
FMOD uses a 'closest sound to the listener' method to determine what should be heard in this case.

See Also

- EventSystem::get3DNumListeners
- EventSystem::set3DListenerAttributes
Firelight Technologies FMOD Ex
**EventSystem::setLanguage**

Globally sets the requested language for future loading of wave banks, where multiple languages are supported. FEV files that are loaded from this point on will load the banks that match with the selected language.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::setLanguage(
    const char *  language
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_SetLanguage(
    FMOD_EVENTSYSTEM *  eventSystem,
    const char *  language
);
```

**Parameters**

*language*

A string containing the requested language.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

This function can be called after EventSystem::load. If a current language related bank is loaded, then **EventGroup::freeEventData** needs to be called to unload it,
followed by a call to this function then EventGroup::loadEventData / EventGroup::getEvent will be needed.
There is no error checking on language strings passed in. If the language does not exist then it will be ignored upon loading wave banks and the previous or default language will be used.
Upper or lower case is not important in this function, FMOD will match the string against the internally stored FEV language list without regard for case.

See Also

- EventSystem::getLanguage
- EventGroup::loadEventData
- EventGroup::freeEventData
- EventGroup::getEvent
Firelight Technologies FMOD Ex
EventSystem::setMediaPath

Specify a base search path for media files so they can be placed somewhere other than the directory of the main executable.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::setMediaPath(
    const char * path
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_SetMediaPath(
    FMOD_EVENTSYSTEM * eventSystem,
    const char * path
);
```

Parameters

`path`

A character string containing a correctly formatted path to load media files from. NOTE: Must contain a trailing slash/backslash if filesystem requires it!

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

FMOD used to add a slash/backslash seperator between the path provided here and the files that it needed to load. This caused inconsistent and inflexible
behaviour so this function has been changed to expect a path that already contains a trailing slash/backslash. FMOD will no longer add any slash/backslash separators between the path specified here and the files that it needs to load. This allows the user to provide the correct separator for the filesystem in use - which may actually be the user's own filesystem which may or may not use a separator at all.

See Also

- **EventSystem::load**
Firelight Technologies FMOD Ex
EventSystem::setPluginPath

Specify a base search path for plugins so they can be placed somewhere other than the directory of the main executable.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::setPluginPath(
    const char * path
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_SetPluginPath(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * path
);
```

Parameters

`path`

A character string containing a correctly formatted path to load plugins from. You can specify 0 or NULL to tell FMOD not to load any plugins.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventSystem::load
Firelight Technologies FMOD Ex
EventSystem::setReverbAmbientProp

Sets a 'background' default reverb environment for the virtual reverb system. This is a reverb preset that will be morphed to if the listener is not within any virtual reverb zones.
By default the ambient reverb is set to 'off'.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::setReverbAmbientProperties(
    FMOD_REVERB_PROPERTIES * props
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_SetReverbAmbientProperties(
    FMOD_EVENTSYSTEM * eventsystem,
    FMOD_REVERB_PROPERTIES * props
);
```

Parameters

`props`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

There is one reverb DSP dedicated to providing a 3D reverb effect. This DSP's properties are a weighted sum of all the contributing virtual reverbs. The default 3d reverb properties specify the reverb properties in the 3D volumes
which has no virtual reverb defined.

**EventSystem::getReverbPreset** and **EventSystem::getReverbPresetByIndex** can be used to retrieve sound designer defined presets, or it can be set programatically.

**See Also**

- **FMOD_REVERB_PROPERTIES**
- **EventSystem::getReverbAmbientProperties**
- **EventSystem::createReverb**
- **EventSystem::getReverbPreset**
- **EventSystem::getReverbPresetByIndex**
Firelight Technologies FMOD Ex
EventSystem::setReverbProperties

Sets parameters for the global reverb environment. Reverb parameters can be set manually, or automatically using the pre-defined presets given in the fmod.h header.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::setReverbProperties(
    const FMOD_REVERB_PROPERTIES * props
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_SetReverbProperties(
    FMOD_EVENTSYSTEM * eventsystem,
    const FMOD_REVERB_PROPERTIES * props
);
```

Parameters

`props`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

`EventSystem::getReverbPreset` and `EventSystem::getReverbPresetByIndex` can be used to retrieve sound designer defined presets, or it can be set programatically.
See Also

- **FMOD_REVERB_PROPERTIES**
- `EventSystem::getReverbProperties`
- `Event::setReverbProperties`
- `Event::getReverbProperties`
- `EventSystem::getReverbPreset`
- `EventSystem::getReverbPresetByIndex`
Firelight Technologies FMOD Ex
**EventSystem::setUserData**

Sets a user value that the EventSystem object will store internally. Can be retrieved with `EventSystem::getUserData`.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::setUserData(
    void * userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_SetUserData(
    FMOD_EVENTSYSTEM * eventsystem,
    void * userdata
);
```

**Parameters**

`userdata`

Address of user data that the user wishes stored within the EventSystem object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This function is primarily used in case the user wishes to 'attach' data to an FMOD object. It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using `EventSystem::getUserData` would help in the identification of the object.

See Also

- `EventSystem::getUserData`
Firelight Technologies FMOD Ex
EventSystem::unload

Unloads all loaded event projects.

C++ Syntax

```cpp
FMOD_RESULT EventSystem::unload();
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_Unload(FMOD_EVENTSYSTEM * eventSystem);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This unloads all loaded projects. If you want to unload a single project, use `EventProject::release`.

See Also

- `EventSystem::load`
- `EventProject::release`
Firelight Technologies FMOD Ex
EventSystem::unloadFSB

De-register a preloaded FSB from the event system.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::unloadFSB(
    const char * filename,
    int streaminstance
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_UnloadFSB(
    FMOD_EVENTSYSTEM * eventsystem,
    const char * filename,
    int streaminstance
);
```

**Parameters**

*filename*

FSB filename to unregister.

*streaminstance*

If a streaming fsb has been registered multiple times, this value represents which instance of the filename should be unloaded.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

Any references to this filename by the EventSystem internally will be loaded from disk as normal if this FSB is unregistered as a preloaded FSB.

See Also

- EventSystem::preloadFSB
Firelight Technologies FMOD Ex
**EventSystem::update**

Updates the event system. This should be called once per 'game' tick, or once per frame in your application.

**C++ Syntax**

```cpp
FMOD_RESULT EventSystem::update();
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventSystem_Update(FMOD_EVENTSYSTEM * eventsystem);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [EventSystem::init](#)
Firelight Technologies FMOD Ex
EventProject Interface

Evento::cancelAllLoads EventProject::getEvent
Evento::getEventByProjectID
Evento::getGroup
Evento::getGroupByIndex
Evento::getInfo
Evento::getMemoryInfo
Evento::getNumEvents
Evento::getNumGroups
Evento::getUserData
Evento::loadSampleData
Evento::release
Evento::setUserData
Evento::stopAllEvents
Firelight Technologies FMOD Ex
EventProject::cancelAllLoads

Cancels any queued up asynchronous loads caused by `FMOD_EVENT_NONBLOCKING` flag.

**C++ Syntax**

```cpp
FMOD_RESULT EventProject::cancelAllLoads();
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventProject_CancelAllLoads(FMOD_EVENTPROJECT * eventproject);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.  
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Note, if this function is called, the eventGroup that you are loading may only be half loaded.  
The state of the event group is set to `FMOD_ERR_FILE_DISKEJECTED`. To clear this state and fully load the event group, it may have to be unloaded with `EventGroup::freeEventData` and reloaded again with `EventGroup::loadEventData`.

**See Also**
- EventProject::stopAllEvents
- EventGroup::loadEventData
- EventGroup::freeEventData
- FMOD_EVENT_MODE
Firelight Technologies FMOD Ex
EventProject::getEvent

Retrieve an event object by name.

C++ Syntax

```cpp
FMOD_RESULT EventProject::getEvent(
    const char * name,
    FMOD_EVENT_MODE mode,
    Event ** event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetEvent(
    FMOD_EVENTPROJECT * eventproject,
    const char * name,
    FMOD_EVENT_MODE mode,
    FMOD_EVENT ** event
);
```

Parameters

**name**

The name of an event within this event project. Note: name must include full path including any event group names e.g. "group1/group2/myevent"

**mode**

If not already loaded with EventGroup::loadEventData, this **FMOD_EVENT_MODE** flag will determine if data should be loaded from disk synchronously or asynchronously. It also allows an 'info only' event retrieval (events that cant be played) and can be used to produce an error on disk access.

**event**

Address of a variable to receive the selected event within this event project.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

What is an event?
An event is the leaf of the event group tree. It is the actual sound to be played with complex behaviour designed by the sound designer.

In this case we are retrieving an event from an **event group**, so with the "**filters**"
group we could get the echo event with "echo" as the name parameter.

If the programmer does not know which events are available, the sound designer tool can output a programmer report that lists the event group's events with the appropriate names and indices listed alongside them.

**Note!**
- An event is retrieved from a pool of events (created earlier if FMOD_EVENT_CACHEEVENTS flag was set in `EventSystem::getGroup` / `EventGroup::getGroup`).
- Data may not be loaded from the disk for this event, so this event may trigger disk access. If you wish to pre-emp this use `EventGroup::loadEventData` first.
- The pool of events has a size determined by the 'max playbacks' property in the FMOD Designer tool in the event's property sheet.
- The pointer to will be getting will be a pointer to one of these event instances.
- If you call this function more times than there are event instances, then an event handle may be stolen, or may fail. This behaviour also determined by the sound designer. The behaviour may be to steal the oldest event in the pool, steal the quietest event in the pool, or simply fail this `getEvent` and return null as the event handle.

**See Also**

- `EventProject::getEventByProjectID`
- `EventGroup::getEventByIndex`
- `EventSystem::getGroup`
- `EventGroup::getGroup`
- `FMOD_EVENT_MODE`
Firelight Technologies FMOD Ex
EventProject::getEventByProjectID

Retrieve an event handle by a project unique id.

C++ Syntax

```cpp
FMOD_RESULT EventProject::getEventByProjectID(
    unsigned int  projectid,
    FMOD_EVENT_MODE  mode,
    Event **  event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetEventByProjectID(
    FMOD_EVENTPROJECT *  eventproject,
    unsigned int  projectid,
    FMOD_EVENT_MODE  mode,
    FMOD_EVENT **  event
);
```

Parameters

`projectid`

The project id of an event within this event project. Unique ids can be found in the programmer report generated when the project is built, and the C header.

`mode`

If not already loaded with EventGroup::loadEventData, this `FMOD_EVENT_MODE` flag will determine if data should be loaded from disk synchronously or asynchronously. It also allows an 'info only' event retrieval (events that can't be played) and can be used to produce an error on disk access.

`event`

Address of a variable to receive the selected event within this event project.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

What is an event?
An event is the leaf of the event group tree. It is the actual sound to be played with complex behaviour designed by the sound designer.

In this case we are retrieving an event from an event group, so with the "filters"
group we could get the echo event with "echo" as the name parameter.

If the programmer does not know which events are available, the sound designer tool can output a programmer report that lists the event group's events with the appropriate names and indices listed alongside them.

**Note!**
- An event is retrieved from a pool of events (created earlier if FMOD_EVENT_CACHEEVENTS flag was set in `EventSystem::getGroup / EventGroup::getGroup`).
- Data may not be loaded from the disk for this event, so this event may trigger disk access. If you wish to pre-emp this use EventGroup::loadEventData first.
- The pool of events has a size determined by the 'max playbacks' property in the FMOD Designer tool in the event's property sheet.
- The pointer to will be getting will be a pointer to one of these event instances.
- If you call this function more times than there are event instances, then an event handle may be stolen, or may fail. This behaviour also determined by the sound designer. The behaviour may be to steal the oldest event in the pool, steal the quietest event in the pool, or simply fail this getEvent and return null as the event handle.

**See Also**

- EventProject::getEvent
- EventProject::getNumEvents
- EventGroup::getEventByIndex
- EventSystem::getGroup
- EventGroup::getGroup
- FMOD_EVENT_MODE
Firelight Technologies FMOD Ex
EventProject::getGroup

Retrieves an event group object by name.

C++ Syntax

```cpp
FMOD_RESULT EventProject::getGroup(
    const char * name,
    bool cacheevents,
    EventGroup ** group
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetGroup(
    FMOD_EVENTPROJECT * eventproject,
    const char * name,
    FMOD_BOOL cacheevents,
    FMOD_EVENTGROUP ** group
);
```

Parameters

name

The name of an event group that belongs to this event project.

cacheevents

If cacheevents is true then all event instances within this event group will be pre-allocated so that there are no memory allocs when getEvent is called.

group

Address of a variable to receive the selected event group within this event project.

Return Values
If the function succeeds then the return value is \texttt{FMOD\_OK}.
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

\textbf{See Also}

- \texttt{EventProject::getGroupByIndex}
- \texttt{EventGroup::getGroup}
- \texttt{EventGroup::getGroupByIndex}
EventProject::getGroupByIndex

Retrieves an event group object by index.

C++ Syntax

```cpp
FMOD_RESULT EventProject::getGroupByIndex(
    int index,
    bool cacheevents,
    EventGroup ** group
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetGroupByIndex(
    FMOD_EVENTPROJECT * eventproject,
    int index,
    FMOD_BOOL cacheevents,
    FMOD_EVENTGROUP ** group
);
```

Parameters

index

The index of an event group within this event project. Indices are 0 based.

cacheevents

If cacheevents is true then all event instances within this event group will be pre-allocated so that there are no memory allocs when getEvent is called.

group

Address of a variable to receive the selected event group within this event project.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- `EventProject::getNumGroups`
- `EventProject::getGroup`
- `EventGroup::getGroup`
- `EventGroup::getGroupByIndex`
Firelight Technologies FMOD Ex
EventProject::getInfo

Retrieve information about this event project.

C++ Syntax

```cpp
FMOD_RESULT EventProject::getInfo(
    FMOD_EVENT_PROJECTINFO * info
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetInfo(
    FMOD_EVENTPROJECT * eventproject,
    FMOD_EVENT_PROJECTINFO * info
);
```

Parameters

`info`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventGroup::getInfo`
- `Event::getInfo`
Firelight Technologies FMOD Ex
EventProject::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```c++
FMOD_RESULT EventProject::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetMemoryInfo(
    FMOD_EVENTPROJECT * eventproject,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

event_memorybits

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

memoryused

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated

`FMOD_MEMORY_USAGE_DETAILS` structure to be filled with detailed
memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. 
If the function fails then the return value will be one of the values defined in the

`FMOD_RESULT` enumeration.

**Remarks**

See `System::getMemoryInfo` for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGE_DETAILS`
- `System::getMemoryInfo`
Firelight Technologies FMOD Ex
EventProject::getNumEvents

Returns the total number of events for this project only.

C++ Syntax

```cpp
FMOD_RESULT EventProject::getNumEvents(
    int * numevents
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetNumEvents(
    FMOD_EVENTPROJECT * eventproject,
    int * numevents
);
```

Parameters

`numevents`

Address of a variable to receive the number of events in this project.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [EventProject::getEventByProjectID](#)
Firelight Technologies FMOD Ex
EventProject::getNumGroups

Retrieves the number of event groups stored within this event project.

C++ Syntax

```cpp
FMOD_RESULT EventProject::getNumGroups(
    int * numgroups
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetNumGroups(
    FMOD_EVENTPROJECT * eventproject,
    int * numgroups
);
```

Parameters

`numgroups`

Address of a variable to receive the number of groups within this event project.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventProject::getGroupByIndex`
Firelight Technologies FMOD Ex
EventProject::getUserData

Retrieves the user value that that was set by calling the EventProject::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT EventProject::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_GetUserData(
    FMOD_EVENTPROJECT * eventproject,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the EventProject::setUserData function.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventProject::setUserData
Firelight Technologies FMOD Ex
EventProject::loadSampleData

Load sample data for multiple events/eventgroups at once in an optimized way.

C++ Syntax

```cpp
FMOD_RESULT EventProject::loadSampleData(
    int * eventid_array,
    int  sizeof_eventid_array,
    char ** groupname_array,
    int  sizeof_groupname_array,
    FMOD_EVENT_MODE  eventmode
);```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_LoadSampleData(
    FMOD_EVENTPROJECT * eventproject,
    int * eventid_array,
    int  sizeof_eventid_array,
    char ** groupname_array,
    int  sizeof_groupname_array,
    FMOD_EVENT_MODE  eventmode
);
```

Parameters

`eventid_array`

Address of an array of project IDs of events to load sample data for. Note: These are project IDs, not system IDs.

`sizeof_eventid_array`

Number of project IDs in the eventid_array.

`groupname_array`

Address of an array of eventgroup names to load sample data for. Note: eventgroup names must be specified using their full path.
sizeof_groupname_array

Number of eventgroup names in the groupname_array.

eventmode

Use either FMOD_EVENT_DEFAULT for blocking behaviour or FMOD_EVENT_NONBLOCKING for non-blocking behaviour.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This is the fastest way of loading a disparate assortment of events and eventgroups. It's faster than multiple calls to EventGroup::loadEventData because it globs together the information on what FSB subsounds are required and then performs one big FSB read to load them in all at the same time with minimal seeking.

Use this function when you want to load a pre-defined set of events and/or eventgroups in one big chunk e.g. at the start of a level. If you require more "random access" or "on-the-fly" loading of events/eventgroups then EventGroup::loadEventData may be a better fit.

Example :

```c
const int SIZEOF_EVENTID_ARRAY = 4;
int eventid_array[SIZEOF_EVENTID_ARRAY] =
{
    EVENT_EXAMPLES_FEATUREDEMONSTRATION_BASICSIMPLEEVENT,
    EVENT_EXAMPLES_FEATUREDEMONSTRATION_SEQUENCINGANDSTITCHING_PROGRAMMERSELECTED,
    EVENT_EXAMPLES_ADVANCEDTECHNIQUES_CAR,
    EVENT_EXAMPLES_ADVANCEDTECHNIQUES_LOOPBASEDMUSIC,
};
```
const int SIZEOF_GROUPNAME_ARRAY = 2;
char *groupname_array[SIZEOF_GROUPNAME_ARRAY] =
{
    "FeatureDemonstration/Randomization",
    "FeatureDemonstration/Effects",
};
result = eventproject->loadSampleData(eventid_array, SIZEOF_EVENTID_ARRAY,

See Also

- EventGroup::loadEventData
- EventGroup::freeEventData
- FMOD_EVENT_MODE
Firelight Technologies FMOD Ex
EventProject::release

Release this event project and all the events/eventgroups that it contains.

C++ Syntax

```cpp
FMOD_RESULT EventProject::release();
```

C Syntax

```c
FMOD_RESULT FMOD_EventProject_Release(FMOD_EVENTPROJECT * eventproj);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Note: This function may block momentarily if there are asynchronous loads currently queued up due to `FMOD_NONBLOCKING` / `FMOD_EVENT_NONBLOCKING` use.

See Also

- `EventSystem::getProject`
- `EventSystem::getProjectByIndex`
- `EventSystem::unload`
Firelight Technologies FMOD Ex
EventProject::setUserData

Sets a user value that the EventProject object will store internally. Can be retrieved with EventProject::getUserData.

C++ Syntax

```
FMOD_RESULT EventProject::setUserData(
    void * userdata
);
```

C Syntax

```
FMOD_RESULT FMOD_EventProject_SetUserData(
    FMOD_EVENTPROJECT * eventproject,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the EventProject object.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using EventProject::getUserData would help in the identification of the object.

See Also

- EventProject::getUserData
Firelight Technologies FMOD Ex
**EventProject::stopAllEvents**

Stop all events in this project.

**C++ Syntax**

```cpp
FMOD_RESULT EventProject::stopAllEvents(
    bool immediate
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventProject_StopAllEvents(
    FMOD_EVENTPROJECT * eventproject,
    FMOD_BOOL    immediate
);
```

**Parameters**

*immediate*

A value to pass as the 'immediate' parameter to Event::stop

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

This function simply iterates through all events in the project and calls Event::stop on them.
See Also

- Event::stop
Firelight Technologies FMOD Ex
EventGroup Interface

EventGroup::freeEventData EventGroup::getEvent
EventGroup::getEventByIndex
EventGroup::getGroup
EventGroup::getGroupByIndex
EventGroup::getInfo
EventGroup::getMemoryInfo
EventGroup::getNumEvents
EventGroup::getNumGroups
EventGroup::getNumProperties
EventGroup::getParentGroup
EventGroup::getParentProject
EventGroup::getProperty
EventGroup::getPropertyByIndex
EventGroup::getState
EventGroup::getUserData
EventGroup::loadEventData
EventGroup::setUserData
Firelight Technologies FMOD Ex
EventGroup::freeEventData

Free the wave data and event instances for an EventGroup and all subgroups under it or for just a single specified event.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::freeEventData(
    Event * event,
    bool waituntilready
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_FreeEventData(
    FMOD_EVENTGROUP * eventgroup,
    FMOD_EVENT * event,
    FMOD_BOOL waituntilready
);
```

Parameters

`event`

Single event to free resources for. Specify 0 or NULL to ignore. Note: The specified event must exist in this EventGroup, not in a child of this EventGroup.

`waituntilready`

If TRUE, this function will block until all pending asynchronous loads have completed before freeing the event data. If FALSE, this function will return `FMOD_ERR_NOTREADY` if any asynchronous loads are pending and it will NOT free any event data. Default = TRUE.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the
**Remarks**

If no event is specified then resources for all events in this EventGroup, and all EventGroups within this EventGroup, will be freed. If an event is specified then all resources for all instances of that event only will be freed.

NOTE: This function does not completely remove events from memory, it simply frees any event instances and wave data allocated by them. To completely remove events from memory, use EventSystem::unload.

Use waituntilready = false in time-critical situations to avoid blocking the main thread. Note that if **FMOD_ERR_NOTREADY** is returned from this function then no event data was actually freed - you will need to call this function again until it succeeds.

Note: Do not call this function from an event callback!

**See Also**

- EventGroup::loadEventData
- EventSystem::getGroup
- Event::release
Firelight Technologies FMOD Ex
EventGroup::getEvent

Retrieve an event object by name.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getEvent(
    const char * name,
    FMOD_EVENT_MODE mode,
    Event ** event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetEvent(
    FMOD_EVENTGROUP * eventgroup,
    const char * name,
    FMOD_EVENT_MODE mode,
    FMOD_EVENT ** event
);
```

Parameters

name

The name of an event within this event group.

mode

If not already loaded with EventGroup::loadEventData, this
FMOD_EVENT_MODE flag will determine if data should be loaded from disk
synchronously or asynchronously. It also allows an 'info only' event retrieval
(events that cant be played) and can be used to produce an error on disk access.

event

Address of a variable to receive the selected event within this event group.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

What is an event?
An event is the leaf of the event group tree. It is the actual sound to be played with complex behaviour designed by the sound designer.

In this case we are retrieving an event from an event group, so with the "filters"
group we could get the echo event with "echo" as the name parameter.

If the programmer does not know which events are available, the sound designer tool can output a programmer report that lists the event group's events with the appropriate names and indices listed alongside them.

**Note!**
- An event is retrieved from a pool of events (created earlier if FMOD_EVENT_CACHEEVENTS flag was set in `EventSystem::getGroup` / `EventGroup::getGroup`).
- Data may not be loaded from the disk for this event, so this event may trigger disk access. If you wish to pre-emp this use EventGroup::loadEventData first.
- The pool of events has a size determined by the 'max playbacks' property in the FMOD Designer tool in the event's property sheet.
- The pointer to will be getting will be a pointer to one of these event instances.
- If you call this function more times than there are event instances, then an invent handle may be stolen, or may fail. This behaviour also determined by the sound designer. The behaviour may be to steal the oldest event in the pool, steal the quietest event in the pool, or simply fail this getEvent and return null as the event handle.

**See Also**

- `EventGroup::getEventByIndex`
- `EventSystem::getGroup`
- `EventGroup::getGroup`
- `FMOD_EVENT_MODE`
Firelight Technologies FMOD Ex
EventGroup::getEventByIndex

Retrieve an event object by index for this group.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getEventByIndex(
    int index,
    FMOD_EVENT_MODE mode,
    Event ** event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetEventByIndex(
    FMOD_EVENTGROUP * eventgroup,
    int index,
    FMOD_EVENT_MODE mode,
    FMOD_EVENT ** event
);
```

Parameters

**index**

The index of an event within this event sub-group. Indices are 0 based.

**mode**

If not already loaded with EventGroup::loadEventData, this 
**FMOD_EVENT_MODE** flag will determine if data should be loaded from disk synchronously or asynchronously. It also allows an 'info only' event retrieval (events that can't be played) and can be used to produce an error on disk access.

**event**

Address of a variable to receive the selected event within this event group.
**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

**What is an event?**
An event is the leaf of the event group tree. It is the actual sound to be played with complex behaviour designed by the sound designer.

In this case we are retrieving an event from an **event group**, so with the "**filters**"
group we could get the echo event with 2 as the index parameter.

If the programmer does not know which events are available or which event index matches which event name, the sound designer tool can output a programmer report that lists the event group's events with the appropriate names and indices listed alongside them. The only benefit of retrieving an object by index is that it is slightly faster to do so than to retrieve it by name.

**Note!**
- An event is retrieved from a pool of events (created earlier if FMOD_EVENT_CACHEEVENTS flag was set in EventSystem::getGroup / EventGroup::getGroup).
- Data may not be loaded from the disk for this event, so this event may trigger disk access. If you wish to pre-emp this use EventGroup::loadEventData first.
- The pool of events has a size determined by the 'max playbacks' property in the FMOD Designer tool in the event's property sheet.
- The pointer to will be getting will be a pointer to one of these event instances.
- If you call this function more times than there are event instances, then an invent handle may be stolen, or may fail. This behaviour also determined by the sound designer. The behaviour may be to steal the oldest event in the pool, steal the quietest event in the pool, or simply fail this getEvent and return null as the event handle.

**See Also**

- EventGroup::getEvent
- EventGroup::getNumEvents
- FMOD_EVENT_MODE
Firelight Technologies FMOD Ex
# EventGroup::getGroup

Retrieves an event group's sub-group object by name.

**C++ Syntax**

```cpp
FMOD_RESULT EventGroup::getGroup(
    const char * name,
    bool cacheevents,
    EventGroup ** group
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventGroup_GetGroup(
    FMOD_EVENTGROUP * eventgroup,
    const char * name,
    FMOD_BOOL cacheevents,
    FMOD_EVENTGROUP ** group
);
```

**Parameters**

*name*

The name of an event sub-group that belongs to this event group.

*cacheevents*

If cacheevents is true then all event instances within this event group will be pre-allocated so that there are no memory allocs when getEvent is called.

*group*

Address of a variable to receive the selected event sub-group within this event group.

**Return Values**
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

**What is an event group?**

An event group is a "folder" that stores events or sub-folders. With these folders a hierarchical tree can be built to store events in a more logical manner.

In this case we are retrieving an event group from another event group, so if this event group object was "examples" we could then get the event group "filters" with "filters" as the name parameter.
In this example "filters" is the only sub-group below "examples" so no other sub-groups are available here.

If the programmer does not know which sub-groups are available or which sub-group index matches which sub-group name, the sound designer tool can output a programmer report that lists the group's sub-groups with the appropriate names and indices listed alongside them. The only benefit of retrieving an object by index is that it is slightly faster to do so than to retrieve it by name.

**See Also**

- EventSystem::getGroup
- EventGroup::getGroupByIndex
Firelight Technologies FMOD Ex
EventGroup::getGroupByIndex

Retrieves an event group's sub-group object by index.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getGroupByIndex(
    int index,
    bool cacheevents,
    EventGroup ** group
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetGroupByIndex(
    FMOD_EVENTGROUP * eventgroup,
    int index,
    FMOD_BOOL cacheevents,
    FMOD_EVENTGROUP ** group
);
```

Parameters

`index`

The index of an event sub-group within this event group. Indices are 0 based.

`cacheevents`

If cacheevents is true then all event instances within this event group will be pre-allocated so that there are no memory allocs when getEvent is called.

`group`

Address of a variable to receive the selected event sub-group within this event group.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

**What is an event group?**
An event group is a "folder" that stores events or sub-folders. With these folders a hierarchical tree can be built to store events in a more logical manner.

![Diagram of FMOD Designer with event groups](image)

In this case we are retrieving an event group from another **event group**, so if this event group object was "**examples**" we could then get the event group "**filters**" with 0 as the index parameter.
In this example "filters" is the only sub-group below "examples" so no other sub-groups are available here.

If the programmer does not know which groups are available or which event group index matches which group name, the sound designer tool can output a programmer report that lists the group's sub-groups with the appropriate names and indices listed alongside them. The only benefit of retrieving an object by index is that it is slightly faster to do so than to retrieve it by name.

See Also

- EventSystem::getGroup
- EventGroup::getGroup
- EventGroup::getNumGroups
Firelight Technologies FMOD Ex
EventGroup::getInfo

Retrieve information about this event group.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getInfo(
    int * index,
    char ** name
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetInfo(
    FMOD_EVENTGROUP * eventgroup,
    int * index,
    char ** name
);
```

Parameters

index

Address of a variable to receive the event group index.

name

Address of a variable to receive the event group name.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also
- EventParameter::getInfo
- Event::getInfo
Firelight Technologies FMOD Ex
EventGroup::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetMemoryInfo(
    FMOD_EVENTGROUP * eventgroup,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits

Memory usage bits for FMOD Ex. See FMOD_MEMBITS.

event_memorybits

Memory usage bits for FMOD Event System. See FMOD_EVENT_MEMBITS.

memoryused

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated

**FMOD_MEMORY_USAGE_DETAILS** structure to be filled with detailed
memory usage information about this object.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the

**FMOD_RESULT** enumeration.

**Remarks**

See [System::getMemoryInfo](#) for more details.

**See Also**

- **FMOD_MEMBITS**
- **FMOD_EVENT_MEMBITS**
- **FMOD_MEMORY_USAGE_DETAILS**
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
EventGroup::getNumEvents

Retrieves the number of event events stored within this event group.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getNumEvents(
    int * numevents
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetNumEvents(
    FMOD_EVENTGROUP * eventgroup,
    int * numevents
);
```

Parameters

`numevents`

Address of a variable to receive the number of events within this event group.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventGroup::getIndexByEvent
Firelight Technologies FMOD Ex
EventGroup::getNumGroups

Retrieves the number of event groups stored within this event group.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getNumGroups(
    int * numgroups
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetNumGroups(
    FMOD_EVENTGROUP * eventgroup,
    int * numgroups
);
```

Parameters

`numgroups`

Address of a variable to receive the number of groups within this event group.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [EventGroup::getGroupByIndex](#)
Firelight Technologies FMOD Ex
EventGroup::getNumProperties

Retrieve the number of properties for an event group.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getNumProperties(
    int * numproperties
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetNumProperties(
    FMOD_EVENTGROUP * eventgroup,
    int * numproperties
);
```

Parameters

`numproperties`

Address of a variable to receive the number of properties for this event group.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [EventGroup::getPropertyByIndex](#)
Firelight Technologies FMOD Ex
**EventGroup::getParentGroup**

Retrieves the eventgroup object to which this eventgroup belongs.

**C++ Syntax**

```cpp
FMOD_RESULT EventGroup::getParentGroup(
    EventGroup ** group
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventGroup_GetParentGroup(
    FMOD_EVENTGROUP * eventgroup,
    FMOD_EVENTGROUP ** group
);
```

**Parameters**

*group*

Address of a variable that receives a pointer to the eventgroup's parent eventgroup

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

The top level eventgroup will return a parent of 0 or NULL, as it has no parent.
See Also

- EventGroup::getGroup
- EventGroup::getGroupByIndex
Firelight Technologies FMOD Ex
EventGroup::getParentProject

Retrieves the eventproject object to which this eventgroup belongs.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getParentProject(
    EventProject ** project
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetParentProject(
    FMOD_EVENTGROUP * eventgroup,
    FMOD_EVENTPROJECT ** project
);
```

Parameters

`project`

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventGroup::getGroup
- EventGroup::getGroupByIndex
Firelight Technologies FMOD Ex
EventGroup::getProperty

Retrieve an event group property by name.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getProperty(
    const char * propertyname,
    void * value
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetProperty(
    FMOD_EVENTGROUP * eventgroup,
    const char * propertyname,
    void * value
);
```

Parameters

`propertyname`

Name of the property to retrieve. This is the name that was specified in FMOD Designer.

`value`

Address of a variable to receive the event group property.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- EventGroup::getPropertyByIndex
- EventGroup::getNumProperties
- EventGroup::getEvent
Firelight Technologies FMOD Ex
EventGroup::getPropertyByIndex

Retrieve an event group property by index.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getPropertyByIndex(
    int propertyindex,
    void * value
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup.GetPropertyByIndex(
    FMOD_EVENTGROUP * eventgroup,
    int propertyindex,
    void * value
);
```

Parameters

`propertyindex`

Index of the property to retrieve.

`value`

Address of a variable to receive the event group property.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also
• `EventGroup::getProperty`
• `EventGroup::getNumProperties`
• `EventGroup::getEvent`
Firelight Technologies FMOD Ex
EventGroup::getState

Retrieves the current state of an event group.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getState(
    FMOD_EVENT_STATE * state
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetState(
    FMOD_EVENTGROUP * eventgroup,
    FMOD_EVENT_STATE * state
);
```

Parameters

`state`

Address of a variable that receives the event group's current state.

Return Values

If the function succeeds then the return value is `FMOD_OK`.  
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

When `FMOD_EVENT_STATE_PLAYING` is set, at least one event in the group is playing.  
When `FMOD_EVENT_STATE_LOADING` is set, at least one event in this eventgroup is loading, else no events in this eventgroup are loading.
FMOD_EVENT_STATE_LOADING, FMOD_EVENT_STATE_ERROR or FMOD_EVENT_STATE_READY will be the state after using EventGroup::loadEventData.

Note: The absence of FMOD_EVENT_STATE_LOADING does not imply that all event data is loaded, just that nothing is in the process of loading at this point in time.

See Also

- FMOD_EVENT_STATE
- EventGroup::loadEventData
Firelight Technologies FMOD Ex
EventGroup::getUserData

Retrieves the user value that that was set by calling the EventGroup::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_GetUserData(
    FMOD_EVENTGROUP * eventgroup,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the EventGroup::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventGroup::setUserData
Firelight Technologies FMOD Ex
EventGroup::loadEventData

Loads wave data and allocates event instances for all events within an event group.

**C++ Syntax**

```cpp
FMOD_RESULT EventGroup::loadEventData(
    FMOD_EVENT_RESOURCE resource,
    FMOD_EVENT_MODE mode
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventGroup_LoadEventData(
    FMOD_EVENTGROUP * eventgroup,
    FMOD_EVENT_RESOURCE resource,
    FMOD_EVENT_MODE mode
);
```

**Parameters**

*resource*

Type of wave data to load. Either load samples, streams, or both. See [FMOD_EVENT_RESOURCE](#).

*mode*

Use either [FMOD_EVENT_DEFAULT](#) for blocking behaviour or [FMOD_EVENT_NONBLOCKING](#) for non-blocking behaviour.

**Return Values**

If the function succeeds then the return value is [FMOD_OK](#).
If the function fails then the return value will be one of the values defined in the [FMOD_RESULT](#) enumeration.
Remarks

Use EventGroup::freeEventData to unload wave data for this group. Note that if another event in a different group is still using the wave data, it will not be freed until those events have had freeEventData called on them as well. (On their parent group).

NOTE: Event instances will always be allocated by this function regardless of what parameters are passed in.

See Also

- EventGroup::freeEventData
- EventSystem::getGroup
- FMOD_EVENT_RESOURCE
- FMOD_EVENT_MODE
- EventProject::loadSampleData
Firelight Technologies FMOD Ex
EventGroup::setUserData

Sets a user value that the EventGroup object will store internally. Can be retrieved with EventGroup::getUserData.

C++ Syntax

```cpp
FMOD_RESULT EventGroup::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventGroup_SetUserData(
    FMOD_EVENTGROUP * eventgroup,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the EventGroup object.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using `EventGroup::getUserData` would help in the identification of the object.

**See Also**

- `EventGroup::getUserData`
Firelight Technologies FMOD Ex
Event Interface

Event::get3DAttributes  Event::get3DOcclusion
Event::getCategory
Event::getChannelGroup
Event::getInfo
Event::getMemoryInfo
Event::getMute
Event::getNumParameters
Event::getNumProperties
Event::getParameter
Event::getParameterByIndex
Event::getParentGroup
Event::getPaused
Event::getPitch
Event::getProperty
Event::getPropertyByIndex
Event::getPropertyInfo
Event::getReverbProperties
Event::getState
Event::getUserData
Event::getVolume
Event::release
Event::set3DAttributes
Event::set3DOcclusion
Event::setCallback
Event::setMute
Event::setPaused
Event::setPitch
Event::setProperty
Event::setPropertyByIndex
Event::setReverbProperties
Event::setUserData
Event::setVolume
Event::start
Event::stop
Firelight Technologies FMOD Ex
Event::get3DAttributes

Retrieves the position and velocity of an event.

C++ Syntax

```cpp
FMOD_RESULT Event::get3DAttributes(
    FMOD_VECTOR * position,
    FMOD_VECTOR * velocity,
    FMOD_VECTOR * orientation
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_Get3DAttributes(
    FMOD_EVENT * event,
    FMOD_VECTOR * position,
    FMOD_VECTOR * velocity,
    FMOD_VECTOR * orientation
);
```

Parameters

**position**

Address of a variable that receives the position in 3D space of the event. Optional. Specify 0 to ignore.

**velocity**

Address of a variable that receives the velocity in 'distance units per second' in 3D space of the event. See remarks. Optional. Specify 0 to ignore.

**orientation**

Address of a variable that receives the orientation of the event. Optional. Specify 0 to ignore. Only used for events with sound cones specified.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

A 'distance unit' is specified in the FMOD Designer tool and are the distance units used by the game (i.e. feet, meters, inches, centimeters etc). An event has to be 3D to have its 3d position and velocity set.

See Also

- [Event::set3DAttributes](#)

---

Version 4.44.07 Built on Feb 11, 2013
Firelight Technologies FMOD Ex
Event::get3DOcclusion

Retrieves the the EAX or software based occlusion factors for an event.

C++ Syntax

```cpp
FMOD_RESULT Event::get3DOcclusion(
    float * directocclusion,
    float * reverbocclusion
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_Get3DOcclusion(
    FMOD_EVENT * event,
    float * directocclusion,
    float * reverbocclusion
);
```

Parameters

directocclusion

Address of a variable that receives the occlusion factor for a voice for the direct path. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0. Optional. Specify 0 or NULL to ignore.

reverbocclusion

Address of a variable that receives the occlusion factor for a voice for the reverb mix. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0. Optional. Specify 0 or NULL to ignore.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

With EAX based sound cards, or I3DL2 based hardware accelerated voices, this will attenuate the sound and frequencies. With non EAX or I3DL2 harward accelerated voices, then the volume is attenuated by the directOclusion factor.

See Also

- Event::set3DOcclusion
Firelight Technologies FMOD Ex
**Event::getCategory**

Retrieve an event category that this event belongs to.

**C++ Syntax**

```cpp
FMOD_RESULT Event::getCategory(
    EventCategory ** category
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Event_GetCategory(
    FMOD_EVENT * event,
    FMOD_EVENTCATEGORY ** category
);
```

**Parameters**

*category*

Address of a variable to receive the event category.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- [Event::getInfo](#)
- [EventGroup::getEvent](#)
Firelight Technologies FMOD Ex
Event::getChannelGroup

Retrieves a pointer to a lower level ChannelGroup class, mainly so that the programmer can add a custom DSP effect with ChannelGroup::addDSP.

C++ Syntax

```
FMOD_RESULT Event::getChannelGroup(
    FMOD::ChannelGroup ** channelgroup
);
```

C Syntax

```
FMOD_RESULT FMOD_Event_GetChannelGroup(
    FMOD_EVENT * event,
    FMOD_CHANNELGROUP ** channelgroup
);
```

Parameters

`channelgroup`

Address of a variable to receive a pointer to a low level ChannelGroup class.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

If you get `FMOD_ERR_DSP_NOTFOUND` when using the retrieved channelgroup, then the event system has optimized the channelgroup's DSP unit away to save memory. You can force the channelgroup to contain a DSP unit
with the `FMOD_EVENT_USERDSP` flag.

**See Also**

- `FMOD_RESULT`
- `FMOD_EVENT_MODE`
Firelight Technologies FMOD Ex
Event::getInfo

Retrieves information about the event.

C++ Syntax

```cpp
FMOD_RESULT Event::getInfo(
    int * index,
    char ** name,
    FMOD_EVENT_INFO * info
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetInfo(
    FMOD_EVENT * event,
    int * index,
    char ** name,
    FMOD_EVENT_INFO * info
);
```

Parameters

`index`

Address of a variable to receive the event group's index. Specify 0 or NULL to ignore.

`name`

Address of a variable to receive the event name. Specify 0 or NULL to ignore.

`info`

Address of an `FMOD_EVENT_INFO` structure to receive extended event information. Specify 0 or NULL to ignore.

Return Values
If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

The **FMOD_EVENT_INFO** structure has members that need to be initialized before `Event::getInfo` is called. Always initialize the **FMOD_EVENT_INFO** structure before calling `Event::getInfo`!

**See Also**

- `EventGroup::getEvent`
- `EventGroup::getEventByIndex`
- `EventParameter::getInfo`
- **FMOD_EVENT_INFO**
Firelight Technologies FMOD Ex
Event::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT Event::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetMemoryInfo(
    FMOD_EVENT * event,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

`memorybits`

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

`event_memorybits`

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

`memoryused`

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

`memoryused_details`
Optional. Specify 0 to ignore. Address of a user-allocated `FMOD_MEMORY_USAGE_DETAILS` structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See `System::getMemoryInfo` for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGE_DETAILS`
- `System::getMemoryInfo`
Firelight Technologies FMOD Ex
Event::getMute

Retrieves the muted state of an event.

C++ Syntax

```cpp
FMOD_RESULT Event::getMute(
    bool * mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetMute(
    FMOD_EVENT * event,
    FMOD_BOOL * mute
);
```

Parameters

mute

Address of a variable to receive the muted state of the event.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [Event::setMute](#)
Firelight Technologies FMOD Ex
Event::getNumParameters

Retrieve the number of parameters for an event.

C++ Syntax

```cpp
FMOD_RESULT Event::getNumParameters(
    int * numparameters
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetNumParameters(
    FMOD_EVENT * event,
    int * numparameters
);
```

Parameters

`numparameters`

Address of a variable to receive the number of parameters for this event.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Event::getParameterByIndex`
Firelight Technologies FMOD Ex
Event::getNumProperties

Retrieve the number of properties for an event.

C++ Syntax

```cpp
FMOD_RESULT Event::getNumProperties(
    int * numproperties
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetNumProperties(
    FMOD_EVENT * event,
    int * numproperties
);
```

Parameters

`numproperties`

Address of a variable to receive the number of properties for this event.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Event::getPropertyByIndex
Firelight Technologies FMOD Ex
Event::getParameter

Retrieve an event parameter object by name.

C++ Syntax

```c++
FMOD_RESULT Event::getParameter(
    const char * name,
    EventParameter ** parameter
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetParameter(
    FMOD_EVENT * event,
    const char * name,
    FMOD_EVENTPARAMETER ** parameter
);
```

Parameters

name

The name of an event parameter that belongs to this event.

parameter

Address of a variable to receive the selected event parameter within this event.

Return Values

If the function succeeds then the return value is **FMOD_OK**.

If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks
Each event will have a list of parameters to control the playback behaviour of the event. For example, if a sound designer made a car engine event, one of the parameters might be 'RPM'. If the programmer does not know which parameters are available, the sound designer tool can output a programmer report that lists the event's parameters with the appropriate names and indices listed alongside them.

See Also

- `Event::getParameterByIndex`
- `EventGroup::getEvent`
- `EventGroup::getEventByIndex`
- `FMOD_EVENT_MODE`
Firelight Technologies FMOD Ex
Event::getParameterByIndex

Retrieve an event parameter by index.

C++ Syntax

```cpp
FMOD_RESULT Event::getParameterByIndex(
    int index,
    EventParameter ** parameter
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetParameterByIndex(
    FMOD_EVENT * event,
    int index,
    FMOD_EVENTPARAMETER ** parameter
);
```

Parameters

`index`

The index of an event parameter within this event. Indices are 0 based. Pass -1 to retrieve this event's primary parameter.

`parameter`

Address of a variable to receive the event parameter object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Each event will have a list of parameters to control the playback behaviour of the event. For example, if a sound designer made a car engine event, one of the parameters might be 'RPM'.

If the programmer does not know which parameters are available or which index matches which parameter, the sound designer tool can output a programmer report that lists the event's parameters with the appropriate names and indices listed alongside them.

The only benefit of retrieving a parameter by index is that it is slightly faster to do so than to retrieve it by name.

See Also

- Event::getParameter
- Event::getNumParameters
- EventGroup::getEvent
- EventGroup::getEventByIndex
Firelight Technologies FMOD Ex
Event::getParentGroup

Retrieves the eventgroup object to which this event belongs.

C++ Syntax

```cpp
FMOD_RESULT Event::getParentGroup(
    EventGroup ** group
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetParentGroup(
    FMOD_EVENT * event,
    FMOD_EVENTGROUP ** group
);
```

Parameters

`group`

Address of a variable that receives a pointer to the event's parent eventgroup.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [EventGroup::getEvent](#)
- [EventGroup::getEventByIndex](#)
Firelight Technologies FMOD Ex
Event::getPaused

Retrieves the paused state of an event.

C++ Syntax

```cpp
FMOD_RESULT Event::getPaused(
    bool * paused);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetPaused(
    FMOD_EVENT * event,
    FMOD_BOOL * paused);
```

Parameters

`paused`

Address of a variable to receive the paused state of the event.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `Event::setPaused`
Firelight Technologies FMOD Ex
Event::getPitch

Retrieves the overall pitch of an event.

C++ Syntax

```cpp
FMOD_RESULT Event::getPitch(
    float * pitch,
    FMOD_EVENT_PITCHUNITS units
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetPitch(
    FMOD_EVENT * event,
    float * pitch,
    FMOD_EVENT_PITCHUNITS units
);
```

Parameters

pitch

Address of a variable to receive the current pitch level of the event. 0.0 = normal pitch (default).

units

The desired units for the retrieved pitch value.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- `Event::setPitch`
- `FMOD_EVENT_PITCHUNITS`
Firelight Technologies FMOD Ex
Event::getProperty

Retrieve an event user property by name.

C++ Syntax

```cpp
FMOD_RESULT Event::getProperty(
    const char * propertyname,
    void * value,
    bool this_instance
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetProperty(
    FMOD_EVENT * event,
    const char * propertyname,
    void * value,
    FMOD_BOOL this_instance
);
```

Parameters

`propertyname`

Name of the user property to retrieve. This is the name that was specified in FMOD Designer.

`value`

Address of a variable to receive the event user property.

`this_instance`

If TRUE then retrieve the per-instance user property value, if FALSE then retrieve the parent (FMOD_EVENT_INFOONLY) event's user property value.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

A parent, or **FMOD_EVENT_INFOONLY**, event is the prototype that all its event instances are based on. An event instance obtained by using any of the getEventXXX functions will be initialized with the current property values of its parent event. After an event instance is obtained, its property values may be modified using **Event::setProperty** and **Event::setPropertyByIndex** so that they differ from their parent event's properties. Use the 'this_instance' parameter to specify whether to retrieve the property value of the parent event or the specific event instance.

Note: This function is used to access user properties only. To access built-in event properties use **Event::getPropertyByIndex**.

**See Also**

- **Event::getPropertyByIndex**
- **Event::setProperty**
- **Event::setPropertyByIndex**
- **EventGroup::getEvent**
Firelight Technologies FMOD Ex
Event::getPropertyByIndex

Retrieve an event property by index.

C++ Syntax

```cpp
FMOD_RESULT Event::getPropertyByIndex(
    int propertyindex,
    void * value,
    bool this_instance
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetPropertyByIndex(
    FMOD_EVENT * event,
    int propertyindex,
    void * value,
    FMOD_BOOL this_instance
);
```

Parameters

**propertyindex**

Index of the property to retrieve. See `FMOD_EVENT_PROPERTY` for details.

**value**

Address of a variable to receive the event property.

**this_instance**

If TRUE then retrieve the per-instance property value, if FALSE then retrieve the parent (`FMOD_EVENT_INFOONLY`) event's property value.

Return Values
If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

A parent, or **FMOD_EVENT_INFOONLY**, event is the prototype that all its event instances are based on. An event instance obtained by using any of the getEventXXX functions will be initialized with the current property values of its parent event. After an event instance is obtained, its property values may be modified using **Event::setProperty** and **Event::setPropertyByIndex** so that they differ from their parent event's properties. Use the 'this_instance' parameter to specify whether to retrieve the property value of the parent event or the specific event instance.

To access user properties by index, add **FMOD_EVENTPROPERTY_USER_BASE** to propertyindex e.g. to retrieve user property 2, set propertyindex to **FMOD_EVENTPROPERTY_USER_BASE** + 2.

**See Also**

- **Event::getProperty**
- **Event::getNumProperties**
- **Event::setProperty**
- **Event::setPropertyByIndex**
- **EventGroup::getEvent**
- **FMOD_EVENT_PROPERTY**
Firelight Technologies FMOD Ex
Event::getPropertyInfo

Retrieve name or index information about a user property.

C++ Syntax

```cpp
FMOD_RESULT Event::getPropertyInfo(
    int * propertyindex,
    char ** propertyname,
    FMOD_EVENTPROPERTY_TYPE * type
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetPropertyInfo(
    FMOD_EVENT * event,
    int * propertyindex,
    char ** propertyname,
    FMOD_EVENTPROPERTY_TYPE * type
);
```

Parameters

`propertyindex`

Pointer to a user property index. See remarks for details.

`propertyname`

Pointer to a user property name. See remarks for details.

`type`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Use this function to retrieve a user property's name from its index or, conversely, its index from its name:

If "*propertyname" is non-NULL on entry then this function will set "*propertyindex" to the index of the user property called "*propertyname".
If "*propertyname" is NULL on entry then this function will set "*propertyname" to the name of the user property with index "*propertyindex".

If the specified user property is not found then FMOD_ERR_INVALID_PARAM will be returned.

**Note!** This function works on user properties only. When specifying "propertyindex" on entry, remember that user property indices begin at FMOD_EVENTPROPERTY_USER_BASE.

**Note!** Do not free the "*propertyname" pointer obtained from this function! FMOD owns it.
Firelight Technologies FMOD Ex
Event::getReverbProperties

Retrieves the current reverb properties for this event.

C++ Syntax

```cpp
FMOD_RESULT Event::getReverbProperties(
    FMOD_REVERB_CHANNELPROPERTIES * props
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetReverbProperties(
    FMOD_EVENT * event,
    FMOD_REVERB_CHANNELPROPERTIES * props
);
```

Parameters

`props`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Event::setReverbProperties
- `FMOD_REVERB_CHANNELPROPERTIES`
Firelight Technologies FMOD Ex
Event::getState

Retrieves the current state of an event.

C++ Syntax

```cpp
FMOD_RESULT Event::getState(
    FMOD_EVENT_STATE * state
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetState(
    FMOD_EVENT * event,
    FMOD_EVENT_STATE * state
);
```

Parameters

`state`

Address of a variable that receives the event's current state.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventGroup::loadEventData`
- `EventGroup::getEvent`
- `FMOD_EVENT_STATE`
- `FMOD_EVENT_MODE`
Firelight Technologies FMOD Ex
Event::getUserData

Retrieves the user value that was set by calling the Event::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT Event::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetUserData(
    FMOD_EVENT * event,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the Event::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- Event::setUserData
Firelight Technologies FMOD Ex
Event::getVolume

Retrieves the overall volume of an event.

C++ Syntax

```cpp
FMOD_RESULT Event::getVolume(
    float * volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_GetVolume(
    FMOD_EVENT * event,
    float * volume
);
```

Parameters

`volume`

Address of a variable to receive the current volume level of the event. 0.0 = silent, 1.0 = full volume (default).

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- Event::setVolume
Firelight Technologies FMOD Ex
**Event::release**

Release this event instance back to the event system so it can be reused.

**C++ Syntax**

```cpp
FMOD_RESULT Event::release(
    bool freeeventdata,
    bool waituntilready
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Event_Release(
    FMOD_EVENT * event,
    FMOD_BOOL freeeventdata,
    FMOD_BOOL waituntilready
);
```

**Parameters**

*freeeventdata*

Set this to true to free event data for this event instance.

*waituntilready*

If TRUE, this function will block until all pending asynchronous loads have completed before freeing the event data. If FALSE, this function will return **FMOD_ERR_NOTREADY** if any asynchronous loads are pending and it will NOT free any event data. Default = TRUE.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

Note: This function is currently only for when you've specified >0 value for FMOD_EVENT_LOADINFO.sizeof_instancepool_simple.

When using an event instance pool (by specifying a >0 value in FMOD_EVENT_LOADINFO.sizeof_instancepool_simple) this function becomes very important. Use this function to release event instances back to the instance pool so they can be reused. If you don't release event instances back to the instance pool it may be possible for events with high "Steal priority" to hog the instance pool, not allowing events with low "Steal priority" to be acquired.

When using an event instance pool for events with different "Steal priorities", always release event instances when you're finished with them!

Use waituntilready = false in time-critical situations to avoid blocking the main thread. Note that if FMOD_ERR_NOTREADY is returned from this function then no event data was actually freed - you will need to call this function again until it succeeds.

Note: Do not call this function from an event callback!

See Also

- EventGroup::getEvent
- EventGroup::getEventByIndex
- EventGroup::freeEventData
Firelight Technologies FMOD Ex
Event::set3DAttributes

Sets the 3d position and velocity of an event.

C++ Syntax

```cpp
FMOD_RESULT Event::set3DAttributes(
    const FMOD_VECTOR * position,
    const FMOD_VECTOR * velocity,
    const FMOD_VECTOR * orientation
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_Set3DAttributes(
    FMOD_EVENT * event,
    const FMOD_VECTOR * position,
    const FMOD_VECTOR * velocity,
    const FMOD_VECTOR * orientation
);
```

Parameters

position

Position in 3D space of the event. Specifying 0 / null will ignore this parameter.

velocity

Velocity in 'distance units per second' in 3D space of the event. See remarks. Specifying 0 / null will ignore this parameter.

orientation

Orientation of the event sound cone. Only used if the event has a cone specified to determine cone detection, otherwise just specify 0 / null. Specifying 0 / null will ignore this parameter.
Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks

A 'distance unit' is specified in the FMOD Designer tool and are the distance units used by the game (i.e. feet, meters, inches, centimeters etc). An event has to be 3D to have its 3d position and velocity set.

Before getting an event with 'just fail if quietest' max playbacks behaviour, this function should be called on an EVENT_INFOONLY event to allow the event system to estimate volume. The various getEvent* functions copy the 3D attributes from the EVENT_INFOONLY event to the event that is returned, so it is not necessary to set the 3D attributes again after getting a real event.

See Also

- [Event::get3DAttributes](#)
Firelight Technologies FMOD Ex
Event::set3DOcclusion

Sets the EAX or software based occlusion factors for an event. This function can be called to produce the same audible effects as the FMOD geometry engine, just without the built in polygon processing.

C++ Syntax

```cpp
FMOD_RESULT Event::set3DOcclusion(
    float directocclusion,
    float reverbocclusion
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_Set3DOcclusion(
    FMOD_EVENT * event,
    float directocclusion,
    float reverbocclusion
);
```

Parameters

`directocclusion`

Occlusion factor for a voice for the direct path. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0.

`reverbocclusion`

Occlusion factor for a voice for the reverb mix. 0.0 = not occluded. 1.0 = fully occluded. Default = 0.0.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

This function does not go through and overwrite the channel occlusion factors, it calls `ChannelGroup::set3DOcclusion`. This means that final occlusion values will be affected by both Event occlusion and geometry (if any).

With EAX based sound cards, or I3DL2 based hardware accelerated voices, this will attenuate the sound and frequencies. With non EAX or I3DL2 hardware accelerated voices, then the volume is attenuated by the directOcclusion factor.

See Also

- `Event::get3DOcclusion`
- `Event::setReverbProperties`
- `ChannelGroup::set3DOcclusion`
Firelight Technologies FMOD Ex
Event::setCallback

Sets a callback so that when certain event behaviours happen, they can be caught by the user.

C++ Syntax

```cpp
FMOD_RESULT Event::setCallback(
    FMOD_EVENT_CALLBACK callback,
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_SetCallback(
    FMOD_EVENT * event,
    FMOD_EVENT_CALLBACK callback,
    void * userdata
);
```

Parameters

`callback`

Pointer to a callback to be called by FMOD.

`userdata`

Userdata pointer to be passed to callback.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

wav sync points are supported. These can be created by placing 'markers' in the original source wavs using a tool such as Sound Forge or Cooledit. FMOD_EVENT_CALLBACKTYPE_SYNCPOINT callbacks will be automatically generated when these markers are encountered.

NOTE: If you call this function on an FMOD_EVENT_INFOONLY event, the callback will be set for ALL instances of the event.

See Also

- FMOD_EVENT_CALLBACK
- FMOD_EVENT_CALLBACKTYPE
Firelight Technologies FMOD Ex
**Event::setMute**

Mutes or unmutes an event.

**C++ Syntax**

```cpp
FMOD_RESULT Event::setMute(
    bool mute
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Event_SetMute(
    FMOD_EVENT * event,
    FMOD_BOOL   mute
);
```

**Parameters**

*mute*

Mute state of the event. true = muted, false = unmuted.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

An event can have several hardware/software voices playing under it at once so this function mutes all relevant voices for this event.
See Also

- Event::getMute
Firelight Technologies FMOD Ex
**Event::setPaused**

Pauses or unpauses an event.

**C++ Syntax**

```cpp
FMOD_RESULT Event::setPaused(
    bool  paused
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Event_SetPaused(
    FMOD_EVENT *  event,
    FMOD_BOOL    paused
);
```

**Parameters**

*paused*

Paused state of the event. true = paused, false = unpaused.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.  
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

An event can have several hardware/software voices playing under it at once so this function pauses all relevant voices for this event.
See Also

- `Event::getPaused`
Firelight Technologies FMOD Ex
Event::setPitch

Sets the overall pitch of an event.

C++ Syntax

```cpp
FMOD_RESULT Event::setPitch(
    float pitch,
    FMOD_EVENT_PITCHUNITS units
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_SetPitch(
    FMOD_EVENT * event,
    float pitch,
    FMOD_EVENT_PITCHUNITS units
);
```

Parameters

pitch

Pitch level of the event. 0.0 = normal pitch (default).

units

The units in which the new pitch level is specified.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

Remarks
An event can have several hardware/software voices playing under it at once so this function scales all relevant voice pitches for this event. This function is not to be used unless needed for runtime reasons, as the sound designer will have set the appropriate event pitch level in the FMOD Designer tool.

See Also

- **Event::getPitch**
- **FMOD_EVENT_PITCHUNITS**
Event::setProperty

Set an event user property by name.

C++ Syntax

```cpp
FMOD_RESULT Event::setProperty(
    const char * propertyname,
    void * value,
    bool this_instance
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_SetProperty(
    FMOD_EVENT * event,
    const char * propertyname,
    void * value,
    FMOD_BOOL this_instance
);
```

Parameters

**propertyname**

Name of the user property to set. This is the name that was specified in FMOD Designer.

**value**

Pointer to the new value for this event user property.

**this_instance**

If TRUE then set the per-instance user property value, if FALSE then set the user property value of all event instances and also the parent (FMOD_EVENT_INFOONLY) event.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

A parent, or `FMOD_EVENT_INFOONLY`, event is the prototype that all its event instances are based on. An event instance obtained by using any of the `getEventXXX` functions will be initialized with the current property values of its parent event. After an event instance is obtained, its property values may be modified using `Event::setProperty` and `Event::setPropertyByIndex` so that they differ from their parent event's properties. Use the 'this_instance' parameter to specify whether to set the property value of all event instances and also the parent event or just the specific event instance.

Note: This function is used to access user properties only. To access built-in event properties use `Event::setPropertyByIndex`.

See Also

- `Event::getProperty`
- `Event::setPropertyByIndex`
- `EventGroup::getEvent`
Firelight Technologies FMOD Ex
**Event::setPropertyByIndex**

Set an event property by index.

**C++ Syntax**

```cpp
FMOD_RESULT Event::setPropertyByIndex(
    int propertyindex,
    void * value,
    bool this_instance
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Event_SetPropertyByIndex(
    FMOD_EVENT * event,
    int propertyindex,
    void * value,
    FMOD_BOOL this_instance
);
```

**Parameters**

- **propertyindex**
  
  Index of the property to set. See [FMOD_EVENT_PROPERTY](#) for details.

- **value**
  
  Pointer to the new value for this event property.

- **this_instance**
  
  If TRUE then set the per-instance property value, if FALSE then set the property value of all event instances and also the parent ([FMOD_EVENT_INFOONLY](#)) event.

**Return Values**
If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

A parent, or `FMOD_EVENT_INFOONLY`, event is the prototype that all its event instances are based on. An event instance obtained by using any of the getEventXXX functions will be initialized with the current property values of its parent event. After an event instance is obtained, its property values may be modified using `Event::setProperty` and `Event::setPropertyByIndex` so that they differ from their parent event's properties. Use the 'this_instance' parameter to specify whether to set the property value of all event instances and also the parent event or just the specific event instance.

To access user properties by index, add `FMOD_EVENTPROPERTY_USER_BASE` to propertyindex e.g. to set user property 2, set propertyindex to `FMOD_EVENTPROPERTY_USER_BASE + 2`.

**See Also**

- `Event::getProperty`
- `Event::setProperty`
- `EventGroup::getEvent`
- `FMOD_EVENTPROPERTY`
Firelight Technologies FMOD Ex
Event::setReverbProperties

Sets the event specific reverb properties, including things like wet/dry mix (Room/Direct), and things like obstruction and occlusion properties.

**C++ Syntax**

```cpp
FMOD_RESULT Event::setReverbProperties(
    const FMOD_REVERB_CHANNELPROPERTIES * props
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Event_SetReverbProperties(
    FMOD_EVENT * event,
    const FMOD_REVERB_CHANNELPROPERTIES * props
);
```

**Parameters**

*props*

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This will scale with the value set by the designer in the event properties.

On PlayStation2, the 'Room' parameter is the only parameter supported. The hardware only allows 'on' or 'off', so the reverb will be off when 'Room' is -10000 and on for every other value.
NOTE: This function overrides values set with `Event::set3DOcclusion`. If you need `Event::set3DOcclusion` functionality then factor it into your `FMOD_REVERB_CHANNELPROPERTIES` values.

See Also

- `Event::getReverbProperties`
- `System::setReverbProperties`
- `FMOD_REVERB_CHANNELPROPERTIES`
- `Event::set3DOcclusion`
Firelight Technologies FMOD Ex
Event::setUserData

Sets a user value that the Event object will store internally. Can be retrieved with Event::getUserData.

C++ Syntax

```cpp
FMOD_RESULT Event::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Event_SetUserData(
    FMOD_EVENT * event,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the Event object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object. It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using `Event::getUserData` would help in the identification of the object.

NOTE: If you call this function on an `FMOD_EVENT_INFOONLY` event, the userdata will only be set for the `FMOD_EVENT_INFOONLY` event, not all of the event's instances.

See Also

- `Event::getUserData`
Firelight Technologies FMOD Ex
**Event::setVolume**

Sets the overall volume of an event.

**C++ Syntax**

```cpp
FMOD_RESULT Event::setVolume(
    float volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Event_SetVolume(
    FMOD_EVENT * event,
    float volume
);
```

**Parameters**

`volume`

Volume level of the event. 0.0 = silent, 1.0 = full volume (default).

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

An event can have several hardware/software voices playing under it at once so this function scales all relevant voice volumes for this event. This function is not to be used unless needed for runtime reasons, as the sound designer will have set the appropriate event volume level in the FMOD Designer.
tool.

**See Also**

- `Event::getVolume`
Firelight Technologies FMOD Ex
Event::start

Start this event playing.

C++ Syntax

```cpp
FMOD_RESULT Event::start();
```

C Syntax

```c
FMOD_RESULT FMOD_Event_Start(FMOD_EVENT * event);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Call `Event::stop` to halt playback of an event.

Attempting to use this function on an `FMOD_EVENT_INFOONLY` event will cause an `FMOD_ERR_INVALID_HANDLE` error to be returned.

See Also

- `Event::stop`
- `EventGroup::getEvent`
- `EventGroup::getEventByIndex`
• FMOD_EVENT_MODE
Firelight Technologies FMOD Ex
**Event::stop**

Stop this event playing.

**C++ Syntax**

```cpp
FMOD_RESULT Event::stop(
    bool  immediate
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Event_Stop(
    FMOD_EVENT *  event,
    FMOD_BOOL     immediate
);
```

**Parameters**

*immediate*

Set this to true to force the event to stop immediately, ignoring the "Fadeout time" property.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Attempting to use this function on an `FMOD_EVENT_INFOONLY` event will cause an `FMOD_ERR_INVALID_HANDLE` error to be returned.
See Also

- Event::start
- EventGroup::getEvent
- EventGroup::getEventByIndex
- FMOD_EVENT_MODE
Firelight Technologies FMOD Ex
EventParameter Interface

EventParameter::disableAutomation  EventParameter::getInfo
EventParameter::getMemoryInfo  EventParameter::getRange
EventParameter::getSeekSpeed  EventParameter::getUserData
EventParameter::getValue  EventParameter::getVelocity
EventParameter::keyOff  EventParameter::setSeekSpeed
EventParameter::setUserData  EventParameter::setValue
EventParameter::setVelocity
Firelight Technologies FMOD Ex
EventParameter::disableAutomation

Turn off automation for a parameter. This is only relevant to automatic parameters such as (distance), (event angle), (listener angle) within FMOD Designer.

C++ Syntax

```cpp
FMOD_RESULT EventParameter::disableAutomation(
    bool disable
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventParameter_DisableAutomation(
    FMOD_EVENTPARAMETER * eventparameter,
    FMOD_BOOL disable
);
```

Parameters

`disable`

When true, FMOD stops automatically updating parameters, allowing the user to set it manually. When false, the parameter updates automatically (default behaviour).

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
EventParameter::getInfo

Retrieve information about this event parameter.

C++ Syntax

```cpp
FMOD_RESULT EventParameter::getInfo(
    int * index,
    char ** name
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventParameter_GetInfo(
    FMOD_EVENTPARAMETER * eventparameter,
    int * index,
    char ** name
);
```

Parameters

index

Address of a variable to receive the event parameters index into the parent event.

name

Address of a variable to receive the event parameter name.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks
Mainly used for display purposes, this function returns a pointer to memory containing the event's name. Do not modify or try to free this memory.

See Also

- EventParameter::getRange
Firelight Technologies FMOD Ex
EventParameter::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT EventParameter::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventParameter_GetMemoryInfo(
    FMOD_EVENTPARAMETER * eventparameter,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits
Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

event_memorybits
Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

memoryused
Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated `FMOD_MEMORY_USAGEDETAILS` structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See [System::getMemoryInfo](#) for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGEDETAILS`
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
EventParameter::getRange

Retrieve the minimum and maximum values for this event parameter.

**C++ Syntax**

```cpp
FMOD_RESULT EventParameter::getRange(
    float * rangemin,
    float * rangemax
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventParameter_GetRange(
    FMOD_EVENTPARAMETER * eventparameter,
    float * rangemin,
    float * rangemax
);
```

**Parameters**

*rangemin*

Address of variable to receive the minimum value allowed for this EventParameter.

*rangemax*

Address of variable to receive the maximum value allowed for this EventParameter.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

This parameter is defined by the sound designer, and usually has a logical meaning, such as RPM for a car engine for example.

See Also

- EventParameter::setValue
- EventParameter::getValue
- EventParameter::getInfo
Firelight Technologies FMOD Ex
EventParameter::getSeekSpeed

Receives the seek velocity of an event.

C++ Syntax

```cpp
FMOD_RESULT EventParameter::getSeekSpeed(
    float * value
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventParameter_GetSeekSpeed( 
    FMOD_EVENTPARAMETER * eventparameter, 
    float * value
);
```

Parameters

`value`

Return Values

If the function succeeds then the return value is `FMOD_OK`
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventParameter::setSeekSpeed
Firelight Technologies FMOD Ex
**EventParameter::getUserData**

Retrieves the user value that was set by calling the `EventParameter::setUserData` function.

**C++ Syntax**

```cpp
FMOD_RESULT EventParameter::getUserData(
    void ** userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventParameter_GetUserData(
    FMOD_EVENTPARAMETER * eventparameter,
    void ** userdata
);
```

**Parameters**

`userdata`

Address of a pointer that receives the data specified with the `EventParameter::setUserData` function.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- `EventParameter::setUserData`
Firelight Technologies FMOD Ex
EventParameter::getValue

Retrieve the current value of this parameter.

C++ Syntax

```cpp
FMOD_RESULT EventParameter::getValue(
    float * value
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventParameter_GetValue(
    FMOD_EVENTPARAMETER * eventparameter,
    float * value
);
```

Parameters

`value`

Address of variable to receive the parameter value.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This parameter is defined by the sound designer, and has a minimum and maximum value. It usually has a logical meaning, such as RPM for a car engine for example.
Attempting to use this function on an `FMOD_EVENT_INFOONLY` event will cause an `FMOD_ERR_INVALID_HANDLE` error to be returned.

**See Also**

- `EventParameter::setValue`
- `EventParameter::getRange`
- `EventParameter::getInfo`
Firelight Technologies FMOD Ex
EventParameter::getVelocity

Receives the velocity of an event.

C++ Syntax

```cpp
FMOD_RESULT EventParameter::getVelocity(
    float * value
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventParameter_GetVelocity(
    FMOD_EVENTPARAMETER * eventparameter,
    float * value
);
```

Parameters

value

Return Values

If the function succeeds then the return value is `FMOD_OK`
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventParameter::setVelocity
Firelight Technologies FMOD Ex
**EventParameter::keyOff**

Triggers a keyoff on an event parameter that has sustain points in it. If an event parameter is currently sustaining on a sustain point, triggering a keyoff will release it and allow the parameter to continue.

**C++ Syntax**

```cpp
FMOD_RESULT EventParameter::keyOff();
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventParameter_KeyOff(FMOD_EVENTPARAMETER * eventp);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Keyoffs can be triggered in advance of a sustain point being reached, so that they continue past the sustain point ahead of time. Attempting to use this function on an `FMOD_EVENT_INFOONLY` event will cause an `FMOD_ERR_INVALID_HANDLE` error to be returned.

**See Also**

- `EventParameter::setVelocity`
- EventParameter::getVelocity
Firelight Technologies FMOD Ex
**EventParameter::setSeekSpeed**

Sets the seek velocity of a parameter.

**C++ Syntax**

```cpp
FMOD_RESULT EventParameter::setSeekSpeed(
    float value
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventParameter_SetSeekSpeed(
    FMOD_EVENTPARAMETER * eventparameter,
    float value
);
```

**Parameters**

`value`

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

Note that currently setting the velocity of an event parameter will set the velocity for all instances of this event. This value is normally set by the sound designer but may be used if the programmer wishes to vary it. Attempting to use this function on an `FMOD_EVENT_INFOONLY` event will cause an `FMOD_ERR_INVALID_HANDLE` error to be returned.
See Also

- EventParameter::getSeekSpeed
Firelight Technologies FMOD Ex
EventParameter::setUserData

Sets a user value that the EventParameter object will store internally. Can be retrieved with EventParameter::getUserData.

C++ Syntax

```cpp
FMOD_RESULT EventParameter::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventParameter_SetUserData(
    FMOD_EVENTPARAMETER * eventparameter,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the EventParameter object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a parameter, and the user does not know which object it is (if many of these types of objects exist). Using `EventParameter::getUserData` would help in the identification of the object.

**See Also**

- `EventParameter::getUserData`
Firelight Technologies FMOD Ex
**EventParameter::setValue**

Set the 'value' of this parameter.

**C++ Syntax**

```cpp
FMOD_RESULT EventParameter::setValue(
    float value
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventParameter_SetValue(
    FMOD_EVENTPARAMETER * eventparameter,
    float value
);
```

**Parameters**

*value*

Value to set this parameter to. Note! Must lie in the range described by `EventParameter::getRange`.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This parameter is defined by the sound designer, and has a minimum and maximum value. It usually has a logical meaning, such as RPM for a car engine for example.
Attempting to use this function on an **FMOD_EVENT_INFOONLY** event will cause an **FMOD_ERR_INVALID_HANDLE** error to be returned.

**See Also**

- `EventParameter::getValue`
- `EventParameter::getRange`
Firelight Technologies FMOD Ex
EventParameter::setVelocity

Sets the velocity of a parameter. In most cases the velocity of a parameter will be 0, unless it is a time based event.

C++ Syntax

```cpp
FMOD_RESULT EventParameter::setVelocity(
    float value
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventParameter_SetVelocity(
    FMOD_EVENTPARAMETER * eventparameter,
    float value
);
```

Parameters

value

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Note that currently setting the velocity of an event parameter will set the velocity for all instances of this event. This value is normally set by the sound designer but may be used if the programmer wishes to pause or speed up / slow down the parameter movement. Attempting to use this function on an `FMOD_EVENT_INFOONLY` event will
cause an `FMOD_ERR_INVALID_HANDLE` error to be returned.

**See Also**

- `EventParameter::getVelocity`
Firelight Technologies FMOD Ex
EventCategory Interface

EventCategory::getCategory  EventCategory::getCategoryByIndex
EventCategory::getChannelGroup
EventCategory::getEventByIndex
EventCategory::getInfo
EventCategory::getMemoryInfo
EventCategory::getMute
EventCategory::getNumCategories
EventCategory::getNumEvents
EventCategory::getParentCategory
EventCategory::getPaused
EventCategory::getPitch
EventCategory::getUserData
EventCategory::getVolume
EventCategory::setMute
EventCategory::setPaused
EventCategory::setPitch
EventCategory::setUserData
EventCategory::setVolume
EventCategory::stopAllEvents
Firelight Technologies FMOD Ex
EventCategory::getCategory

Retrieve an event category object by name.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getCategory(
    const char * name,
    EventCategory ** category
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetCategory(
    FMOD_EVENTCATEGORY * eventcategory,
    const char * name,
    FMOD_EVENTCATEGORY ** category
);
```

Parameters

- **name**
  The name of an event category that belongs to this event category.

- **category**
  Address of a variable to receive the selected event category within this event category.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

Sub-categories can be retrieved by specifying their full path relative to this category e.g. "cars/racers".

See Also

- EventCategory::getCategoryByIndex
- EventSystem::getCategory
Firelight Technologies FMOD Ex
EventCategory::getCategoryByIndex

Retrieve an event category object by index.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getCategoryByIndex(
    int index,
    EventCategory ** category
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetCategoryByIndex(
    FMOD_EVENTCATEGORY * eventcategory,
    int index,
    FMOD_EVENTCATEGORY ** category
);
```

Parameters

index

The index of an event category within this event category. Indices are 0 based.

category

Address of a variable to receive the event category object.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the
**FMOD_RESULT** enumeration.

See Also
• EventCategory::getCategory
• EventSystem::getCategory
Firelight Technologies FMOD Ex
EventCategory::getChannelGroup

Retrieves a pointer to a lower level ChannelGroup class, mainly so that the programmer can add a custom DSP effect with ChannelGroup::addDSP.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getChannelGroup(
    FMOD::ChannelGroup ** channelgroup
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetChannelGroup(
    FMOD_EVENTCATEGORY * eventcategory,
    FMOD_CHANNELGROUP ** channelgroup
);
```

Parameters

channelgroup

Address of a variable to receive a pointer to a low level ChannelGroup class.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Firelight Technologies FMOD Ex
EventCategory::getEventByIndex

Retrieve an event object by index.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getEventByIndex(
    int index,
    FMOD_EVENT_MODE mode,
    Event ** event
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetEventByIndex(
    FMOD_EVENTCATEGORY * eventcategory,
    int index,
    FMOD_EVENT_MODE mode,
    FMOD_EVENT ** event
);
```

Parameters

index

The index of an event within this event category. Indices are 0 based.

mode

If not already loaded with EventGroup::loadEventData, this 
`FMOD_EVENT_MODE` flag will determine if data should be loaded from disk synchronously or asynchronously. It also allows an 'info only' event retrieval (events that can't be played) and can be used to produce an error on disk access.

event

Address of a variable to receive the event object.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventSystem::getCategory`
- `EventCategory::getNumEvents`
- `FMOD_EVENT_MODE`
Firelight Technologies FMOD Ex
EventCategory::getInfo

Retrieve information about this event category.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getInfo(
    int * index,
    char ** name
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetInfo(
    FMOD_EVENTCATEGORY * eventcategory,
    int * index,
    char ** name
);
```

Parameters

`index`

Address of a variable to receive the event category’s index.

`name`

Address of a variable to receive the event category’s name.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks
Mainly used for display purposes, this function returns a pointer to memory containing the event category's name. Do not modify or try to free this memory.

See Also

- EventSystem::getCategory
Firelight Technologies FMOD Ex
EventCategory::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```
FMOD_RESULT EventCategory::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```
FMOD_RESULT FMOD_EventCategory_GetMemoryInfo(
    FMOD_EVENTCATEGORY * eventcategory,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

`memorybits`
Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

`event_memorybits`
Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

`memoryused`
Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

`memoryused_details`
Optional. Specify 0 to ignore. Address of a user-allocated
**FMOD_MEMORY_USAGE_DETAILS** structure to be filled with detailed
memory usage information about this object.

### Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the
**FMOD_RESULT** enumeration.

### Remarks

See [System::getMemoryInfo](#) for more details.

### See Also

- **FMOD_MEMBITS**
- **FMOD_EVENT_MEMBITS**
- **FMOD_MEMORY_USAGE_DETAILS**
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
EventCategory::getMute

Retrieves the mute state of an event category.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getMute(
    bool * mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetMute(
    FMOD_EVENTCATEGORY * eventcategory,
    FMOD_BOOL * mute
);
```

Parameters

`mute`

Address of a variable to receive the mute state of the event category.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventCategory::setMute
Firelight Technologies FMOD Ex
EventCategory::getNumCategories

Retrieve the number of sub-categories below this event category.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getNumCategories(
    int * numcategories
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetNumCategories(
    FMOD_EVENTCATEGORY * eventcategory,
    int * numcategories
);
```

Parameters

`numcategories`

Address of a variable to receive the number of categories in this category.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventCategory::getCategory`
- `EventSystem::getCategory`
Firelight Technologies FMOD Ex
EventCategory::getNumEvents

Retrieve the number of events within this event category.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getNumEvents(
    int * numevents
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetNumEvents(
    FMOD_EVENTCATEGORY * eventcategory,
    int * numevents
);
```

Parameters

numevents

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventSystem::getCategory
- EventCategory::getEventByIndex
Firelight Technologies FMOD Ex
EventCategory::getParentCategory

Retrieves the eventcategory object to which this eventcategory belongs.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getParentCategory(
    EventCategory ** category
);```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetParentCategory(
    FMOD_EVENTCATEGORY * eventcategory,
    FMOD_EVENTCATEGORY ** category
);```

Parameters

category

Address of a variable that receives a pointer to the eventcategory's parent eventcategory

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventCategory::getCategory
- EventSystem::getCategory
Firelight Technologies FMOD Ex
EventCategory::getPaused

Retrieves the paused state of an event category.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getPaused(
    bool * paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetPaused(
    FMOD_EVENTCATEGORY * eventcategory,
    FMOD_BOOL * paused
);
```

Parameters

`paused`

Address of a variable to receive the paused state of the event category.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventCategory::setPaused
Firelight Technologies FMOD Ex
EventCategory::getPitch

Retrieves the overall pitch of an event category.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getPitch(
    float * pitch,
    FMOD_EVENT_PITCHUNITS units
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetPitch(
    FMOD_EVENTCATEGORY * eventcategory,
    float * pitch,
    FMOD_EVENT_PITCHUNITS units
);
```

Parameters

pitch

Address of a variable to receive the current pitch level of the event category. 0.0 = normal pitch (default).

units

The desired units for the retrieved pitch value.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
See Also

- EventCategory::setPitch
- FMOD_EVENT_PITCHUNITS
Firelight Technologies FMOD Ex
EventCategory::getUserData

Retrieves the user value that that was set by calling the EventCategory::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetUserData(
    FMOD_EVENTCATEGORY * eventcategory,
    void ** userdata
);
```

Parameters

userdata

Address of a pointer that receives the data specified with the EventCategory::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventCategory::setUserData
Firelight Technologies FMOD Ex
EventCategory::getVolume

Retrieves the overall volume of an event category.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::getVolume(
    float * volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_GetVolume(
    FMOD_EVENTCATEGORY * eventcategory,
    float * volume
);
```

Parameters

`volume`

Address of a variable to receive the current volume level of the event category. 0.0 = silent, 1.0 = full volume (default).

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventCategory::setVolume
Firelight Technologies FMOD Ex
EventCategory::setMute

Pauses or unpauses an event category for runtime reasons.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::setMute(
    bool mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_SetMute(
    FMOD_EVENTCATEGORY * eventcategory,
    FMOD_BOOL   mute
);
```

Parameters

`mute`

Mute the event category. true = muted, false = unmuted.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventCategory::getMute
Firelight Technologies FMOD Ex
EventCategory::setPaused

Pauses or unpauses an event category for runtime reasons.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::setPaused(
    bool paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_SetPaused(
    FMOD_EVENTCATEGORY * eventcategory,
    FMOD_BOOL paused
);
```

Parameters

`paused`

Paused state of the event category. true = paused, false = unpause.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventCategory::getPaused
Firelight Technologies FMOD Ex
EventCategory::setPitch

Sets the overall pitch of an event category.

**C++ Syntax**

```cpp
FMOD_RESULT EventCategory::setPitch(
    float pitch,
    FMOD_EVENT_PITCHUNITS units
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventCategory_SetPitch(
    FMOD_EVENTCATEGORY * eventcategory,
    float pitch,
    FMOD_EVENT_PITCHUNITS units
);
```

**Parameters**

- **pitch**
  
  Pitch level of the event category. 0.0 = normal pitch (default).

- **units**

  The units in which the new pitch level is specified.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.

If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**
- EventCategory::getPitch
- FMOD_EVENT_PITCHUNITS
Firelight Technologies FMOD Ex
EventCategory::setUserData

Sets a user value that the EventCategory object will store internally. Can be retrieved with EventCategory::getUserData.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_SetUserData(
    FMOD_EVENTCATEGORY * eventcategory,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the EventCategory object.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using `EventCategory::getUserData` would help in the identification of the object.

See Also

- `EventCategory::getUserData`
Firelight Technologies FMOD Ex
EventCategory::setVolume

Sets the overall volume of an event category.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::setVolume(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_SetVolume(
    FMOD_EVENTCATEGORY * eventcategory,
    float volume
);
```

Parameters

`volume`

Volume level of the event category. 0.0 = silent, 1.0 = full volume (default).

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

An event category can have several events playing under it at once so this function scales all relevant event volumes for this event category. This function is not to be used unless needed for runtime reasons, as the sound designer will have set the appropriate event category volume level in the FMOD
Designer tool.

**See Also**

- [EventCategory::getVolume](#)
Firelight Technologies FMOD Ex
EventCategory::stopAllEvents

Stops all events in this category and subcategories.

C++ Syntax

```cpp
FMOD_RESULT EventCategory::stopAllEvents();
```

C Syntax

```c
FMOD_RESULT FMOD_EventCategory_StopAllEvents(FMOD_EVENTCATEGORY * e
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Firelight Technologies FMOD Ex
EventReverb Interface

EventReverb::get3DAttributes  EventReverb::getActive
EventReverb::getMemoryInfo
EventReverb::getProperties
EventReverb::getUserData
EventReverb::release
EventReverb::set3DAttributes
EventReverb::setActive
EventReverb::setProperties
EventReverb::setUserData
Firelight Technologies FMOD Ex
EventReverb::get3DAttributes

Gets the 3D attributes of the event 3D reverb object

C++ Syntax

```cpp
FMOD_RESULT EventReverb::get3DAttributes(
    FMOD_VECTOR * position,
    float * mindistance,
    float * maxdistance
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_Get3DAttributes(
    FMOD_EVENTREVERB * eventreverb,
    FMOD_VECTOR * position,
    float * mindistance,
    float * maxdistance
);
```

Parameters

`position`

pointer to a vector in 3D space where the reverb is centred

`mindistance`

radius within which the reverb has full effect

`maxdistance`

radius outside of which the reverb has zero effect

Return Values

If the function succeeds then the return value is `FMOD_OK`. 
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

\section*{Remarks}

The 3D reverb object is a sphere having 3D attributes (position, minimum distance, maximum distance) and reverb properties. The properties and 3D attributes of all reverb objects collectively determine, along with the listener's position, the settings of and input gains into a single 3D reverb DSP. Please note that this only applies to software channels. When the listener is within the sphere of effect of one or more 3d reverbs, the listener's 3D reverb properties are a weighted combination of such 3d reverbs. When the listener is outside all of the reverbs, the 3D reverb setting is set to the default ambient reverb setting.

\section*{See Also}

- \texttt{EventReverb::set3DAttributes}
Firelight Technologies FMOD Ex
EventReverb::getActive

Retrieves the active state of the reverb object.

C++ Syntax

```cpp
FMOD_RESULT EventReverb::getActive(
    bool * active
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_GetActive(
    FMOD_EVENTREVERB * eventreverb,
    FMOD_BOOL * active
);
```

Parameters

active

Address of a variable to receive the current active state of the reverb object.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventReverb::setActive
- EventSystem::createReverb
Firelight Technologies FMOD Ex
EventReverb::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT EventReverb::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_GetMemoryInfo(
    FMOD_EVENTREVERB * eventreverb,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);```

Parameters

`memorybits`

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

`event_memorybits`

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

`memoryused`

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

`memoryused_details`
Optional. Specify 0 to ignore. Address of a user-allocated \texttt{FMOD_MEMORY_USAGE_DETAILS} structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is \texttt{FMOD_OK}. If the function fails then the return value will be one of the values defined in the \texttt{FMOD_RESULT} enumeration.

**Remarks**

See \texttt{System::getMemoryInfo} for more details.

**See Also**

- \texttt{FMOD_MEMBITS}
- \texttt{FMOD_EVENT_MEMBITS}
- \texttt{FMOD_MEMORY_USAGE_DETAILS}
- \texttt{System::getMemoryInfo}
EventReverb::getProperties

Retrieves the current reverb properties for this event 3d reverb object.

C++ Syntax

```cpp
FMOD_RESULT EventReverb::getProperties(
    FMOD_REVERB_PROPERTIES * props
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_GetProperties(
    FMOD_EVENTREVERB * eventreverb,
    FMOD_REVERB_PROPERTIES * props
);
```

Parameters

`props`

Address of a variable to receive the `FMOD_REVERB_PROPERTIES` information.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventReverb::setProperties`
- `FMOD_REVERB_PROPERTIES`
Firelight Technologies FMOD Ex
EventReverb::getUserData

Retrieves the user value that that was set by calling the EventReverb::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT EventReverb::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_GetUserData(
    FMOD_EVENTREVERB * eventreverb,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the EventReverb::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventReverb::setUserData
Firelight Technologies FMOD Ex
EventReverb::release

Release memory for an event reverb object.

C++ Syntax

```cpp
FMOD_RESULT EventReverb::release();
```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_Release(FMOD_EVENTREVERB * eventreverb)
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This will release this event reverb object.

See Also

- `EventSystem::getReverbPreset`
- `EventSystem::getReverbPresetByIndex`
Firelight Technologies FMOD Ex
EventReverb::set3DAttributes

Sets the 3D attributes of the event 3D reverb object

C++ Syntax

```cpp
FMOD_RESULT EventReverb::set3DAttributes(
    const FMOD_VECTOR * position,
    float mindistance,
    float maxdistance
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_Set3DAttributes(
    FMOD_EVENTREVERB * eventreverb,
    const FMOD_VECTOR * position,
    float mindistance,
    float maxdistance
);
```

Parameters

**position**

pointer to a vector in 3D space where the reverb is centred

**mindistance**

radius within which the reverb has full effect

**maxdistance**

radius outside of which the reverb has zero effect

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.

\textbf{Remarks}

The 3D reverb object is a sphere having 3D attributes (position, minimum distance, maximum distance) and reverb properties. The properties and 3D attributes of all reverb objects collectively determine, along with the listener's position, the settings of and input gains into a single 3D reverb DSP. Please note that this only applies to software channels. When the listener is within the sphere of effect of one or more 3d reverbs, the listener's 3D reverb properties are a weighted combination of such 3d reverbs. When the listener is outside all of the reverbs, the 3D reverb setting is set to the default ambient reverb setting.

\textbf{See Also}

- \texttt{EventReverb::get3DAttributes}
Firelight Technologies FMOD Ex
EventReverb::setActive

Disables or enables a reverb object so that it does or does not contribute to the 3d scene.

C++ Syntax

```cpp
FMOD_RESULT EventReverb::setActive(
    bool active
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_SetActive(
    FMOD_EVENTREVERB * eventreverb,
    FMOD_BOOL active
);
```

Parameters

`active`

true = active, false = not active. Default = true.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventReverb::getActive
- EventSystem::createReverb
Firelight Technologies FMOD Ex
**EventReverb::setProperties**

Sets the reverb properties for this event 3d reverb object.

**C++ Syntax**

```cpp
FMOD_RESULT EventReverb::setProperties(
    const FMOD_REVERB_PROPERTIES * props
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventReverb_SetProperties(
    FMOD_EVENTREVERB * eventreverb,
    const FMOD_REVERB_PROPERTIES * props
);
```

**Parameters**

`props`

Pointer to a `FMOD_REVERB_PROPERTIES` structure definition.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

`EventSystem::getReverbPreset` and `EventSystem::getReverbPresetByIndex` can be used to retrieve sound designer defined presets, or it can be set programmatically.
See Also

- EventReverb::getProperties
- FMOD_REVERB_PROPERTIES
- EventSystem::getReverbPreset
- EventSystem::getReverbPresetByIndex
- EventSystem::createReverb
Firelight Technologies FMOD Ex
EventReverb::setUserData

Sets a user value that the EventReverb object will store internally. Can be retrieved with EventReverb::getUserData.

C++ Syntax

```cpp
FMOD_RESULT EventReverb::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventReverb_SetUserData(
    FMOD_EVENTREVERB * eventreverb,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the EventReverb object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object. It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using EventReverb::getUserData would help in the identification of the object.

**See Also**

- EventReverb::getUserData
Firelight Technologies FMOD Ex
EventQueue Interface

EventQueue::add EventQueue::clear
EventQueue::dump
EventQueue::excludeDuckingCategory
EventQueue::findFirstEntry
EventQueue::findNextEntry
EventQueue::getMemoryInfo
EventQueue::getPaused
EventQueue::getUserData
EventQueue::includeDuckingCategory
EventQueue::release
EventQueue::remove
EventQueue::removeHead
EventQueue::setCallback
EventQueue::setPaused
EventQueue::setUserData
Firelight Technologies FMOD Ex
EventQueue::add

Add an EventQueueEntry to this EventQueue.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::add(
    EventQueueEntry * entry,
    bool allow_duplicates
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_Add(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_EVENTQUEUEENTRY * entry,
    FMOD_BOOL allow_duplicates
);
```

Parameters

`entry`

The EventQueueEntry to add to this EventQueue

`allow_duplicates`

If TRUE, multiple entries for the same event will be allowed. If FALSE, only one entry for any given event will be allowed at one time - if you try to add an entry for an event that is already represented in the queue then this function will fail.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- `EventSystem::createEventQueueEntry`
- `EventQueue::remove`
Firelight Technologies FMOD Ex
EventQueue::clear

Remove all EventQueueEntries from this EventQueue. This function also releases the memory for all EventQueueEntries.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::clear(
    bool stopallevents
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_Clear(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_BOOL stopallevents
);
```

Parameters

`stopallevents`

If TRUE, stop all playing events that were started by this queue. If FALSE, any events played from this queue will continue to play to completion after this queue has been cleared.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventQueue::remove
• EventQueue::add
Firelight Technologies FMOD Ex
EventQueue::dump

Dump information about this EventQueue to the debug log.

C++ Syntax

FMOD_RESULT EventQueue::dump();

C Syntax

FMOD_RESULT FMOD_EventQueue_Dump(FMOD_EVENTQUEUE * eventqueue);

Parameters

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Firelight Technologies FMOD Ex
EventQueue::excludeDuckingCategory

Exclude an EventCategory from ducking when events are playing from this EventQueue.

**C++ Syntax**

```cpp
FMOD_RESULT EventQueue::excludeDuckingCategory(
    EventCategory * category
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventQueue_ExcludeDuckingCategory(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_EVENTCATEGORY * category
);
```

**Parameters**

*category*

The category to exclude from ducking when events are playing from this queue

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

NOTE: This function is currently unimplemented.
See Also

- EventQueue::includeDuckingCategory
Firelight Technologies FMOD Ex
EventQueue::findFirstEntry

Retrieve a pointer to the first EventQueueEntry in this EventQueue. Use this function in conjunction with EventQueue::findNextEntry to iterate through the entries in this EventQueue.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::findFirstEntry(
    EventQueueEntry ** entry
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_FindFirstEntry(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_EVENTQUEUEENTRY ** entry
);
```

Parameters

`entry`

Address of a pointer to receive a pointer to an EventQueueEntry object

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- EventQueue::findNextEntry
Firelight Technologies FMOD Ex
EventQueue::findNextEntry

Retrieve a pointer to the next EventQueueEntry in this EventQueue. Use this function in conjunction with EventQueue::findFirstEntry to iterate through the entries in this EventQueue.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::findNextEntry(
    EventQueueEntry ** entry
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_FindNextEntry(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_EVENTQUEUEENTRY ** entry
);
```

Parameters

`entry`

Address of a pointer to receive a pointer to an EventQueueEntry object. When this function is called, entry must point to a valid EventQueueEntry. Use EventQueue::findFirstEntry to retrieve a valid EventQueueEntry.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also
• EventQueue::findFirstEntry
Firelight Technologies FMOD Ex
EventQueue::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_GetMemoryInfo(
    FMOD_EVENTQUEUE * eventqueue,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits
Memory usage bits for FMOD Ex. See FMOD_MEMBITS.

event_memorybits
Memory usage bits for FMOD Event System. See FMOD_EVENT_MEMBITS.

memoryused
Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated 
**FMOD_MEMORY_USAGE_DETAILS** structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

See [System::getMemoryInfo](#) for more details.

**See Also**

- [FMOD_MEMBITS](#)
- [FMOD_EVENT_MEMBITS](#)
- [FMOD_MEMORY_USAGE_DETAILS](#)
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
EventQueue::getPaused

Get the paused state of this EventQueue.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::getPaused(
    bool * paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_GetPaused(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_BOOL * paused
);
```

Parameters

*paused*

Address of a bool to receive the paused state of this EventQueue.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventQueue::setPaused`
Firelight Technologies FMOD Ex
EventQueue::getUserData

Retrieves the user value that that was set by calling the EventQueue::setUserData function.

C++ Syntax

```
FMOD_RESULT EventQueue::getUserData(
    void ** userdata
);
```

C Syntax

```
FMOD_RESULT FMOD_EventQueue_GetUserData(
    FMOD_EVENTQUEUE * eventqueue,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the EventQueue::setUserData function.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventQueue::setUserData
Firelight Technologies FMOD Ex
EventQueue::includeDuckingCategory

Include an EventCategory to duck when events are playing from this EventQueue.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::includeDuckingCategory(
    EventCategory * category,
    float ducked_volume,
    float unducked_volume,
    unsigned int duck_time,
    unsigned int unduck_time
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_IncludeDuckingCategory(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_EVENTCATEGORY * category,
    float ducked_volume,
    float unducked_volume,
    unsigned int duck_time,
    unsigned int unduck_time
);
```

Parameters

category

The EventCategory to duck when events are playing from this queue

ducked_volume

The volume to duck the category down to (0.0 -> 1.0)

unducked_volume

The volume to unduck the category to when no events are playing from this queue (0.0 -> 1.0)
duck_time

The time in milliseconds over which the category's current volume will be faded down towards the specified ducked_volume

unduck_time

The time in milliseconds over which the category's current volume will be faded up towards the specified unducked_volume

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

Use this function implement "ducking". For example, you may want to temporarily bring down the volume of your background music when a character is talking. You can use this function to specify that the "music" category be ducked by a specified amount whenever something is playing on this EventQueue. Use the "duck_time" and "unduck_time" to specify how fast you want the category to duck and un-duck e.g. shorter "duck_time" will make the background music duck aggressively as soon as someone talks; longer "unduck_time" will keep the music low for a little while after the talking has stopped.

See Also

- EventQueue::excludeDuckingCategory
Firelight Technologies FMOD Ex
EventQueue::release

Release memory for an EventQueue object and all EventQueueEntry objects that it contains.

**C++ Syntax**

```cpp
FMOD_RESULT EventQueue::release();
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventQueue_Release(FMOD_EVENTQUEUE * eventqueue);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

This function will stop all playing events in the queue immediately and call `EventQueueEntry::release` on all EventQueueEntries before releasing the EventQueue itself.

**See Also**

- `EventSystem::createEventQueue`
- `EventQueueEntry::release`
Firelight Technologies FMOD Ex
EventQueue::remove

Remove the specified EventQueueEntry from this EventQueue. This function also releases the memory for the specified EventQueueEntry.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::remove(
    EventQueueEntry * entry
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_Remove(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_EVENTQUEUEENTRY * entry
);
```

Parameters

`entry`

The EventQueueEntry to remove from this EventQueue.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- EventQueue::removeHead
- EventQueue::add
Firelight Technologies FMOD Ex
EventQueue::removeHead

Remove the first EventQueueEntry in this EventQueue. This function also releases the memory for the EventQueueEntry.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::removeHead();
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_RemoveHead(FMOD_EVENTQUEUE * eventqueue)
```

Parameters

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [EventQueue::remove](#)
- [EventQueue::add](#)
Firelight Technologies FMOD Ex
EventQueue::setCallback

Sets a callback so that when certain event queue behaviours happen, they can be caught by the user.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::setCallback(
    FMOD_EVENTQUEUE_CALLBACK callback,
    void * callbackuserdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_SetCallback(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_EVENTQUEUE_CALLBACK callback,
    void * callbackuserdata
);
```

Parameters

`callback`

Pointer to a callback to be called by FMOD.

`callbackuserdata`

User data pointer to be passed to callback.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
See Also

- **FMOD_EVENTQUEUE_CALLBACK**
- **FMOD_EVENTQUEUE_CALLBACKTYPE**
Firelight Technologies FMOD Ex
EventQueue::setPaused

Pause or unpause this EventQueue. When an EventQueue is paused, any events currently playing from this queue will be paused and no new events will be played.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::setPaused(
    bool paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_SetPaused(
    FMOD_EVENTQUEUE * eventqueue,
    FMOD_BOOL paused
);
```

Parameters

*paused*

If TRUE, this EventQueue will be paused. If FALSE, this EventQueue will be unpaued.

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [EventQueue::getPaused](#)
Firelight Technologies FMOD Ex
EventQueue::setUserData

Sets a user value that the EventQueue object will store internally. Can be retrieved with EventQueue::getUserData.

C++ Syntax

```cpp
FMOD_RESULT EventQueue::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueue_SetUserData(
    FMOD_EVENTQUEUE * eventqueue,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the EventQueue object.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a
parameter, and the user does not know which object it is (if many of these types of objects exist). Using EventQueue::getUserData would help in the identification of the object.

See Also

- EventQueue::getUserData
Firelight Technologies FMOD Ex
EventQueueEntry Interface

**getCrossfadeTime**  **getDelayTime**
**getExpiryTime**
**getInfoOnlyEvent**
**getInterrupt**
**getMemoryInfo**
**getPriority**
**getRealEvent**
**getEventData**
**release**
**setCrossfadeTime**
**setDelayTime**
**setExpiryTime**
**setInterrupt**
**setPriority**
**setUserData**
Firelight Technologies FMOD Ex
EventQueueEntry::getCrossfadeTime

Retrieve the crossfade time for this event queue entry.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::getCrossfadeTime(
    int * crossfade
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_GetCrossfadeTime(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    int * crossfade
);
```

Parameters

`crossfade`

Address of a variable to receive the crossfade time of this event queue entry

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventQueueEntry::setCrossfadeTime`
Firelight Technologies FMOD Ex
EventQueueEntry::getDelayTime

Retrieve the delay time for this event queue entry.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::getDelayTime(
    unsigned int * delay
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_GetDelayTime(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    unsigned int * delay
);
```

Parameters

`delay`

Address of a variable to receive the delay time (in milliseconds) of this event queue entry

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventQueueEntry::setDelayTime`
Firelight Technologies FMOD Ex
EventQueueEntry::getExpiryTime

Retrieve the expiry time for this event queue entry.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::getExpiryTime(
   unsigned int * expirytime
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_GetExpiryTime(
   FMOD_EVENTQUEUEENTRY * eventqueueentry,
   unsigned int * expirytime
);
```

Parameters

`expirytime`

Address of a variable to receive the expiry time (in milliseconds) of this event queue entry

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventQueueEntry::setExpiryTime`
Firelight Technologies FMOD Ex
EventQueueEntry::getInfoOnlyEvent

Retrieve a handle to the event that is associated with this event queue entry. Note: This handle will be the same FMOD_EVENT_INFOONLY event handle that was passed to EventSystem::createEventQueueEntry.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::getInfoOnlyEvent(
    Event ** infoonlyevent
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_GetInfoOnlyEvent(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    FMOD_EVENT ** infoonlyevent
);
```

Parameters

`infoonlyevent`

Address of an event handle that receives the FMOD_EVENT_INFOONLY event handle associated with this event queue entry

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventQueueEntry::getRealEvent
Firelight Technologies FMOD Ex
EventQueueEntry::getInterrupt

Retrieve the interrupt flag for this event queue entry.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::getInterrupt(
    bool * interrupt
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_GetInterrupt(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    FMOD_BOOL * interrupt
);
```

Parameters

interrupt

Address of a variable to receive the interrupt flag of this event queue entry

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventQueueEntry::getInterrupt
Firelight Technologies FMOD Ex
EventQueueEntry::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_GetMemoryInfo(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

**memorybits**

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

**event_memorybits**

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

**memoryused**

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

**memoryused_details**
Optional. Specify 0 to ignore. Address of a user-allocated \textbf{FMOD\_MEMORY\_USAGE\_DETAILS} structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is \textbf{FMOD\_OK}.
If the function fails then the return value will be one of the values defined in the \textbf{FMOD\_RESULT} enumeration.

**Remarks**

See \textbf{System\::getMemoryInfo} for more details.

**See Also**

- \textbf{FMOD\_MEMBITS}
- \textbf{FMOD\_EVENT\_MEMBITS}
- \textbf{FMOD\_MEMORY\_USAGE\_DETAILS}
- \textbf{System\::getMemoryInfo}
EventQueueEntry::getPriority

Get the priority of this event queue entry.

**C++ Syntax**

```cpp
FMOD_RESULT EventQueueEntry::getPriority(
  unsigned char * priority
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventQueueEntry_GetPriority(
  FMOD_EVENTQUEUEENTRY * eventqueueentry,
  unsigned char * priority
);
```

**Parameters**

*priority*

Address of a variable to receive the priority of this event queue entry

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- EventQueueEntry::setPriority
Firelight Technologies FMOD Ex
EventQueueEntry::getRealEvent

Retrieve a handle to the playing event that is associated with this event queue entry. This handle will be a valid event instance handle of the playing event, not an `FMOD_EVENT_INFOONLY` handle. If the event is not currently playing, 0 will be returned.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::getRealEvent(
    Event ** realevent
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_GetRealEvent(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    FMOD_EVENT ** realevent
);
```

Parameters

`realevent`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventQueueEntry::getInfoOnlyEvent`
Firelight Technologies FMOD Ex
EventQueueEntry::getUserData

Retrieves the user value that that was set by calling the EventQueueEntry::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::getUserData(
    void ** userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_GetUserData(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    void ** userdata
);
```

Parameters

`userdata`

Address of a pointer that receives the data specified with the EventQueueEntry::setUserData function.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

See Also

- EventQueueEntry::setUserData
Firelight Technologies FMOD Ex
**EventQueueEntry::release**

Release this event queue entry and all associated memory. If this event queue entry is in an event queue, it will be removed from the event queue before it is released.

**C++ Syntax**

```cpp
FMOD_RESULT EventQueueEntry::release();
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventQueueEntry_Release(FMOD_EVENTQUEUEENTRY * eventqueueentry);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- `EventSystem::createEventQueueEntry`
- `EventQueue::add`
Firelight Technologies FMOD Ex
EventQueueEntry::setCrossfadeTime

Set a crossfade time for this event queue entry. NOTE: This function is currently unimplemented.

**C++ Syntax**

```cpp
FMOD_RESULT EventQueueEntry::setCrossfadeTime(
    int crossfade
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_EventQueueEntry_SetCrossfadeTime(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    int crossfade
);
```

**Parameters**

*crossfade*

Crossfade time in milliseconds

**Return Values**

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**

- `EventQueueEntry::getCrossfadeTime`
Firelight Technologies FMOD Ex
EventQueueEntry::setDelayTime

Set a delay time for this event queue entry. Setting a delay time for an entry allows you to delay event playback when an entry reaches the head of the queue.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::setDelayTime(
    unsigned int delay
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_SetDelayTime(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    unsigned int delay
);
```

Parameters

`delay`

Time in milliseconds to delay event playback when this entry reaches the head of the queue. Example: 1500 = event playback will start 1.5 seconds after this entry reaches the head of the queue

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also
• EventQueueEntry::getDelayTime
Firelight Technologies FMOD Ex
EventQueueEntry::setExpiryTime

Set an expiry time for this event queue entry. If this entry sits in the queue for longer than the expiry time without playing then it will expire and be removed from the queue.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::setExpiryTime(
    unsigned int expirytime
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_SetExpiryTime(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    unsigned int expirytime
);
```

Parameters

`expirytime`

Expiry time in milliseconds. Example: 2000 = entry will be removed if it hasn't played within 2 seconds

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- EventQueueEntry::getExpiryTime
● EventQueue::add
Firelight Technologies FMOD Ex
EventQueueEntry::setInterrupt

Set the interrupt flag for this event queue entry. If an entry with the interrupt flag set is added to a queue then it will stop the currently playing event and play immediately.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::setInterrupt(
    bool interrupt
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_SetInterrupt(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    FMOD_BOOL interrupt
);
```

Parameters

`interrupt`

If TRUE, this entry will interrupt the queue when added. If FALSE, this entry will be added to the queue as normal

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventQueueEntry::getInterrupt`
Firelight Technologies FMOD Ex
EventQueueEntry::setPriority

Set the priority of this event queue entry. This determines where this entry will be inserted when it is added to an event queue. Call this function before adding this entry to an event queue.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::setPriority(
    unsigned char priority
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_SetPriority(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    unsigned char priority
);
```

Parameters

`priority`

The priority of this entry 0 = lowest = end of the queue, 255 = highest = start of the queue

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- `EventQueueEntry::getPriority`
• `EventQueue::add`
Firelight Technologies FMOD Ex
EventQueueEntry::setUserData

Sets a user value that the EventQueueEntry object will store internally. Can be retrieved with EventQueueEntry::getUserData.

C++ Syntax

```cpp
FMOD_RESULT EventQueueEntry::setUserData(
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventQueueEntry_SetUserData(
    FMOD_EVENTQUEUEENTRY * eventqueueentry,
    void * userdata
);
```

Parameters

`userdata`

Address of user data that the user wishes stored within the EventQueueEntry object.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function is primarily used in case the user wishes to 'attach' data to an FMOD object.
It can be useful if an FMOD callback passes an object of this type as a parameter, and the user does not know which object it is (if many of these types of objects exist). Using `EventQueueEntry::getUserData` would help in the identification of the object.

**See Also**

- `EventQueueEntry::getUserData`
Firelight Technologies FMOD Ex
MusicSystem Interface

MusicSystem::freeSoundData MusicSystem::getCues
MusicSystem::getInfo
MusicSystem::getMemoryInfo
MusicSystem::getMute
MusicSystem::getNextCue
MusicSystem::getNextParameter
MusicSystem::getParameterValue
MusicSystem::getParameters
MusicSystem::getPaused
MusicSystem::getReverbProperties
MusicSystem::getVolume
MusicSystem::loadSoundData
MusicSystem::prepareCue
MusicSystem::promptCue
MusicSystem::reset
MusicSystem::setCallback
MusicSystem::setMute
MusicSystem::setParameterValue
MusicSystem::setPaused
MusicSystem::setReverbProperties
MusicSystem::setVolume
Firelight Technologies FMOD Ex
MusicSystem::freeSoundData

Frees sound data for the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::freeSoundData(
    bool waituntilready
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_FreeSoundData(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_BOOL waituntilready
);
```

Parameters

`waituntilready`

If TRUE, this function will block until all pending asynchronous loads have completed before freeing the sound data. If FALSE, this function will return `FMOD_ERR_NOTREADY` if any asynchronous loads are pending and it will NOT free any sound data. Default = TRUE.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Note: This function does not completely remove music data from memory, it
simply frees any sounds allocated by the music system. To completely remove music data from memory, use EventSystem::unload. Use waituntilready = false in time-critical situations to avoid blocking the main thread. Note that if FMOD_ERR_NOTREADY is returned from this function then no event data was actually freed - you will need to call this function again until it succeeds.

Note: Do not call this function from a music system callback!

See Also

- MusicSystem::loadSoundData
- EventSystem::unload
Firelight Technologies FMOD Ex
MusicSystem::getCues

Gets a music iterator from the music system pointing to the first cue matching the filter string.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getCues(
    FMOD_MUSIC_ITERATOR * it,
    const char * filter
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetCues(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_ITERATOR * it,
    const char * filter
);
```

Parameters

- `it`
  Address of an iterator to receive the result.

- `filter`
  The filter string used to match cue names. The system will look for an exact match to the cue name.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
Remarks

If the filter string is null the iterator will point to the first cue of all cues sorted alphabetically. If there isn't a match the iterator will hold a null value.

See Also

- `FMOD_MUSIC_ITERATOR`
- `MusicSystem::getNextCue`
Firelight Technologies FMOD Ex
MusicSystem::getInfo

Retrieves information about the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getInfo(
    FMOD_MUSIC_INFO * info
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetInfo(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_INFO * info
);
```

Parameters

`info`

Address of an `FMOD_MUSIC_INFO` structure to receive music system information.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the
`FMOD_RESULT` enumeration.

See Also

- `FMOD_MUSIC_INFO`
Firelight Technologies FMOD Ex
MusicSystem::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetMemoryInfo(
    FMOD_MUSICSYSTEM * musicsystem,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits

Memory usage bits for FMOD Ex. See `FMOD_MEMBITS`.

event_memorybits

Memory usage bits for FMOD Event System. See `FMOD_EVENT_MEMBITS`.

memoryused

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated
**FMOD_MEMORY_USAGE_DETAILS** structure to be filled with detailed
memory usage information about this object.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the
**FMOD_RESULT** enumeration.

**Remarks**

See [System::getMemoryInfo](#) for more details.

**See Also**

- **FMOD_MEMBITS**
- **FMOD_EVENT_MEMBITS**
- **FMOD_MEMORY_USAGE_DETAILS**
- [System::getMemoryInfo](#)
Firelight Technologies FMOD Ex
MusicSystem::getMute

Retrieves the mute state of the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getMute(
    bool * mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetMute(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_BOOL * mute
);
```

Parameters

*mute*

Address of a variable to receive the mute state of the music system.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- MusicSystem::setMute
Firelight Technologies FMOD Ex
MusicSystem::getNextCue

Advances the iterator set by MusicSystem::getCues to the next matching cue in the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getNextCue(
    FMOD_MUSIC_ITERATOR * it
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetNextCue(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_ITERATOR * it
);
```

Parameters

`it`

Address of an iterator to receive the result. Must be the address of an iterator previously given to MusicSystem::getCues.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.

Remarks

If the filter string given to MusicSystem::getCues was null the iterator will point to the next cue of all cues sorted alphabetically. If there isn't a match the iterator
will hold a null value.

See Also

- **FMOD_MUSIC_ITERATOR**
- **MusicSystem::getCues**
Firelight Technologies FMOD Ex
MusicSystem::getNextParameter

Advances the iterator set by `MusicSystem::getParameters` to the next matching parameter in the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getNextParameter(
    FMOD_MUSIC_ITERATOR * it
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetNextParameter(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_ITERATOR * it
);
```

Parameters

`it`

Address of an iterator to receive the result. Must be the address of an iterator previously given to `MusicSystem::getParameters`.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

If the filter string given to `MusicSystem::getParameters` was null the iterator will point to the next parameter of all parameters sorted alphabetically. If there isn't a
match the iterator will hold a null value.

**See Also**

- [FMOD_MUSIC_ITERATOR](#)
- [MusicSystem::getParameters](#)
Firelight Technologies FMOD Ex
**MusicSystem::getParameterValue**

Gets the value of a given music system parameter.

**C++ Syntax**

```cpp
FMOD_RESULT MusicSystem::getParameterValue(
    FMOD_MUSIC_PARAM_ID id,
    float * parameter
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_MusicSystem_GetParameterValue(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_PARAM_ID id,
    float * parameter
);
```

**Parameters**

`id`

The unique identifier for the parameter.

`parameter`

Address of a variable to receive the parameter value.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**
• MusicSystem::setParameterValue
MusicSystem::getParameters

Gets a music iterator from the music system pointing to the first parameter matching the filter string.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getParameters(
    FMOD_MUSIC_ITERATOR * it,
    const char * filter
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetParameters(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_ITERATOR * it,
    const char * filter
);
```

Parameters

`it`

Address of an iterator to receive the result.

`filter`

The filter string used to match parameter names. The system will look for an exact match to the parameter name.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.
**Remarks**

If the filter string is null the iterator will point to the first parameter of all parameters sorted alphabetically. If there isn't a match the iterator will hold a null value.

**See Also**

- [FMOD_MUSIC_ITERATOR](#)
- [MusicSystem::getNextParameter](#)
Firelight Technologies FMOD Ex
MusicSystem::getPaused

Retrieves the paused state of the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getPaused(
    bool * paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetPaused(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_BOOL * paused
);
```

Parameters

`paused`

Address of a variable to receive the paused state of the music system.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [MusicSystem::setPaused](#)
Firelight Technologies FMOD Ex
MusicSystem::getReverbProperties

Gets the reverb properties of the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getReverbProperties(
    FMOD_REVERB_CHANNELPROPERTIES * props
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetReverbProperties(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_REVERB_CHANNELPROPERTIES * props
);
```

Parameters

`props`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

The ConnectionPoint member is always NULL.

See Also

- `MusicSystem::setReverbProperties`
Firelight Technologies FMOD Ex
MusicSystem::getVolume

Retrieves the overall volume of the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::getVolume(
    float * volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_GetVolume(
    FMOD_MUSICSYSTEM * musicsystem,
    float * volume
);
```

Parameters

`volume`

Address of a variable to receive the current volume level of the music system. 0.0 = silent, 1.0 = full volume (default).

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [MusicSystem::setVolume](#)
Firelight Technologies FMOD Ex
MusicSystem::loadSoundData

Loads sound data for the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::loadSoundData(
    FMOD_EVENT_RESOURCE resource,
    FMOD_EVENT_MODE mode
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_LoadSoundData(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_EVENT_RESOURCE resource,
    FMOD_EVENT_MODE mode
);
```

Parameters

resource

Type of data to load. Currently only FMOD_EVENT_RESOURCE_SAMPLES is supported. See FMOD_EVENTRESOURCE.

mode

Only uses FMOD_EVENT_NONBLOCKING flag to determine whether to load asynchronously.

Return Values

If the function succeeds then the return value is FMOD_OK. If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
Remarks

Use `MusicSystem::freeSoundData` to unload music system sound data. Note that if there are events still using the sound data, it will not be freed until those events have had `EventGroup::freeEventData` called on them as well.

See Also

- `MusicSystem::freeSoundData`
- `EventGroup::freeEventData`
- `FMOD_EVENTRESOURCE`
- `FMOD_EVENTMODE`
Firelight Technologies FMOD Ex
**MusicSystem::prepareCue**

Prepares a music prompt for the given cue.

**C++ Syntax**

```cpp
FMOD_RESULT MusicSystem::prepareCue(
    FMOD_MUSIC_CUE_ID id,
    MusicPrompt ** prompt
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_MusicSystem_PrepareCue(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_CUE_ID id,
    FMOD_MUSICPROMPT ** prompt
);
```

**Parameters**

*id*

The unique identifier for the cue.

*prompt*

Address of a variable to receive the prompt object.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**
This function can be used to obtain a MusicPrompt object to trigger and control the lifespan of a cue. **MusicPrompt::begin** and **MusicPrompt::end** can then be called to control the cue.

**Note!** This function allocates memory for the MusicPrompt object. **MusicPrompt::release** must be called to free this memory.

**See Also**

- **MusicSystem::promptCue**
- **MusicPrompt::begin**
- **MusicPrompt::end**
- **MusicPrompt::release**
Firelight Technologies FMOD Ex
MusicSystem::promptCue

Triggers an instantaneous cue.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::promptCue(FMOD_MUSIC_CUE_ID id);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_PromptCue(FMOD_MUSICSYSTEM * musicsystem, FMOD_MUSIC_CUE_ID id);
```

Parameters

`id`

The unique identifier for the cue.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This is a convenience function for triggering cues. The lifespan of the cue within the music system is instantaneous. This function is most useful for triggering musical stings. For a longer musical event `MusicSystem::prepareCue` can be used to obtain a MusicPrompt object to control the lifespan of a cue.
This function is equivalent to calling `MusicSystem::prepareCue` followed immediately by `MusicPrompt::begin` and `MusicPrompt::end`.

See Also

- `MusicSystem::prepareCue`
- `MusicPrompt::begin`
- `MusicPrompt::end`
Firelight Technologies FMOD Ex
**MusicSystem::reset**

Resets the music system.

**C++ Syntax**

```cpp
FMOD_RESULT MusicSystem::reset();
```

**C Syntax**

```c
FMOD_RESULT FMOD_MusicSystem_Reset(FMOD_MUSICSYSTEM * musicsystem);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

This function stops all playing sounds within the music system and clears all active cues.
Firelight Technologies FMOD Ex
MusicSystem::setCallback

Sets a callback so that when certain music behaviours happen, they can be caught by the user.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::setCallback(
    FMOD_MUSIC_CALLBACK callback,
    void * userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_SetCallback(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_CALLBACK callback,
    void * userdata
);
```

Parameters

callback

Pointer to a callback to be called by FMOD.

userdata

User data pointer to be passed to callback.

Return Values

If the function succeeds then the return value is \texttt{FMOD\_OK}.
If the function fails then the return value will be one of the values defined in the \texttt{FMOD\_RESULT} enumeration.
See Also

- `FMOD_MUSIC_CALLBACK`
- `FMOD_MUSIC_CALLBACKTYPE`
Firelight Technologies FMOD Ex
MusicSystem::setMute

Mutes or unmutes the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::setMute(
    bool mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_SetMute(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_BOOL     mute
);
```

Parameters

`mute`

Mute state of the music system. true = muted, false = unmuted.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- MusicSystem::getMute
Firelight Technologies FMOD Ex
**MusicSystem::setParameterValue**

Sets the value of a given music system parameter.

**C++ Syntax**

```cpp
FMOD_RESULT MusicSystem::setParameterValue(
    FMOD_MUSIC_PARAM_ID id,
    float parameter
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_MusicSystem_SetParameterValue(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_MUSIC_PARAM_ID id,
    float parameter
);
```

**Parameters**

*id*

The unique identifier for the parameter.

*parameter*

The value of the parameter to set.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**See Also**
- **MusicSystem::getParameterValue**
Firelight Technologies FMOD Ex
MusicSystem::setPaused

Pauses or unpauses the music system.

**C++ Syntax**

```cpp
FMOD_RESULT MusicSystem::setPaused(
    bool paused
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_MusicSystem_SetPaused(
    FMOD_MUSICSYSTEM * musicsystem,
    FMOD_BOOL paused
);
```

**Parameters**

*paused*

Paused state of the music system. true = paused, false = unpaused.

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**See Also**

- MusicSystem::getPaused
Firelight Technologies FMOD Ex
MusicSystem::setReverbProperties

Sets the reverb properties of the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::setReverbProperties(
    const FMOD_REVERB_CHANNELPROPERTIES * props
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_SetReverbProperties(
    FMOD_MUSICSYSTEM * musicsystem,
    const FMOD_REVERB_CHANNELPROPERTIES * props
);
```

Parameters

`props`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

The ConnectionPoint member will be ignored.

See Also

- MusicSystem::getReverbProperties
Firelight Technologies FMOD Ex
MusicSystem::setVolume

Sets the overall volume of the music system.

C++ Syntax

```cpp
FMOD_RESULT MusicSystem::setVolume(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicSystem_SetVolume(
    FMOD_MUSICSYSTEM * musicsystem,
    float volume
);
```

Parameters

volume

Volume level of the music system. 0.0 = silent, 1.0 = full volume (default).

Return Values

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

See Also

- [MusicSystem::getVolume](#)
Firelight Technologies FMOD Ex
MusicPrompt Interface

MusicPrompt::begin  MusicPrompt::end
MusicPrompt::getMemoryInfo
MusicPrompt::isActive
MusicPrompt::release
Firelight Technologies FMOD Ex
MusicPrompt::begin

Begins a music prompt.

C++ Syntax

```cpp
FMOD_RESULT MusicPrompt::begin();
```

C Syntax

```c
FMOD_RESULT FMOD_MusicPrompt_Begin(FMOD_MUSICPROMPT * musicprompt);
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This function begins the lifespan of the music prompt object. The prompt state becomes active, and any associated actions are performed (e.g. a prompt obtained from `MusicSystem::prepareCue` will activate its associated cue).

This function has no effect if the prompt is already active.

See Also

- `MusicSystem::prepareCue`
Firelight Technologies FMOD Ex
**MusicPrompt::end**

Ends a music prompt.

**C++ Syntax**

```cpp
FMOD_RESULT MusicPrompt::end();
```

**C Syntax**

```c
FMOD_RESULT FMOD_MusicPrompt_End(FMOD_MUSICPROMPT * musicprompt);
```

**Parameters**

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

This function ends the lifespan of the music prompt object. The prompt state becomes inactive, and any associated actions are performed (e.g. a prompt obtained from **MusicSystem::prepareCue** will deactivate its associated cue).

This function has no effect if the prompt is already inactive.

**See Also**

- **MusicSystem::prepareCue**
Firelight Technologies FMOD Ex
MusicPrompt::getMemoryInfo

Retrieve detailed memory usage information about this object.

C++ Syntax

```c++
FMOD_RESULT MusicPrompt::getMemoryInfo(
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicPrompt_GetMemoryInfo(
    FMOD_MUSICPROMPT * musicprompt,
    unsigned int memorybits,
    unsigned int event_memorybits,
    unsigned int * memoryused,
    FMOD_MEMORY_USAGE_DETAILS * memoryused_details
);
```

Parameters

memorybits

Memory usage bits for FMOD Ex. See FMOD_MEMBITS.

event_memorybits

Memory usage bits for FMOD Event System. See FMOD_EVENT_MEMBITS.

memoryused

Optional. Specify 0 to ignore. Address of a variable to receive how much memory is being used by this object given the specified "memorybits" and "event_memorybits".

memoryused_details
Optional. Specify 0 to ignore. Address of a user-allocated `FMOD_MEMORY_USAGE_DETAILS` structure to be filled with detailed memory usage information about this object.

**Return Values**

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

**Remarks**

See `System::getMemoryInfo` for more details.

**See Also**

- `FMOD_MEMBITS`
- `FMOD_EVENT_MEMBITS`
- `FMOD_MEMORY_USAGE_DETAILS`
- `System::getMemoryInfo`
Firelight Technologies FMOD Ex
MusicPrompt::isActive

Retrieves the active state of a music prompt.

C++ Syntax

```cpp
FMOD_RESULT MusicPrompt::isActive(
    bool * active
);
```

C Syntax

```c
FMOD_RESULT FMOD_MusicPrompt_IsActive(
    FMOD_MUSICPROMPT * musicprompt,
    FMOD_BOOL * active
);
```

Parameters

`active`

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

The active state of a music prompt is controlled by `MusicPrompt::begin` and `MusicPrompt::end`.

See Also
• **MusicPrompt::begin**

• **MusicPrompt::end**
Firelight Technologies FMOD Ex
MusicPrompt::release

Releases memory for a MusicPrompt object.

C++ Syntax

```cpp
FMOD_RESULT MusicPrompt::release();
```

C Syntax

```c
FMOD_RESULT FMOD_MusicPrompt_Release(FMOD_MUSICPROMPT * musicprompt
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

This will release this MusicPrompt object, freeing the memory it uses.
Firelight Technologies FMOD Ex
Functions

EventSystem_Create
EventSystem_Create

Factory function to create an EventSystem object. This must be called to create an FMOD System object before you can do anything else.
Use this function to create 1, or multiple instances of FMOD System objects.

C++ Syntax

```cpp
FMOD_RESULT EventSystem_Create(
    EventSystem ** eventsystem
);
```

C Syntax

```c
FMOD_RESULT FMOD_EventSystem_Create(
    FMOD_EVENTSYSTEM ** eventsystem
);
```

Parameters

`eventsystem`

Address of a pointer that receives the new FMOD EventSystem object.

Return Values

If the function succeeds then the return value is `FMOD_OK`.
If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Use `EventSystem::release` to free an eventsystem object.

It is generally recommended to only create one system object. Creating more
than one can lead to excess cpu usage as it will spawn extra software mixer threads.

See Also

- EventSystem::release
Firelight Technologies FMOD Ex
Callbacks

FMOD_EVENTQUEUE_CALLBACK  FMOD_EVENT_CALLBACK  FMOD_MUSIC_CALLBACK
Firelight Technologies FMOD Ex
FMOD_EVENTQUEUE_CALLBACK

EventQueue callback that is called when one of the things listed in FMOD_EVENTQUEUE_CALLBACKTYPE occurs.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_EVENTQUEUE_CALLBACK(
    FMOD_EVENTQUEUE_CALLBACKTYPE type,
    FMOD_EVENTQUEUE * queue,
    FMOD_EVENTQUEUEENTRY * entry,
    void * callbackuserdata
);
```

Parameters

type

Type of callback being issued. See FMOD_EVENTQUEUE_CALLBACKTYPE for the different reasons FMOD will generate a callback for an EventQueue.

queue

The EventQueue that this callback relates to.

entry

The EventQueueEntry that this callback relates to.

callbackuserdata

User specified value set when calling EventQueue::setCallback.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the
**Remarks**

**C++ Users.** Cast **FMOD_EVENTQUEUE** * to **FMOD::EventQueue** * inside the callback and use as normal.  
**C++ Users.** Cast **FMOD_EVENTQUEUEENTRY** * to **FMOD::EventQueueEntry** * inside the callback and use as normal.

To avoid needing to process all messages simply switch on the messages you are interested in.

See **FMOD_EVENTQUEUE_CALLBACKTYPE** for details.

**See Also**

- **FMOD_EVENTQUEUE_CALLBACKTYPE**
- **EventQueue::setCallback**
Firelight Technologies FMOD Ex
**FMOD_EVENT_CALLBACK**

Event callback that is called when one of the events listed in **FMOD_EVENT_CALLBACKTYPE** occurs.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_EVENT_CALLBACK(
    FMOD_EVENT * event,
    FMOD_EVENT_CALLBACKTYPE type,
    void * param1,
    void * param2,
    void * userdata
);
```

**Parameters**

*event*

Event handle in question.

*type*

Type of callback being issued. See **FMOD_EVENT_CALLBACKTYPE** for the different reasons FMOD will generate a callback for an event.

*param1*

Parameter 1 for the event callback. See **FMOD_EVENT_CALLBACKTYPE** for different uses of param1.

*param2*

Parameter 2 for the event callback. See **FMOD_EVENT_CALLBACKTYPE** for different uses of param2.

*userdata*

User specified value set when calling **Event::setCallback**.
Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

C++ Users. Cast `FMOD_EVENT *` to `FMOD::Event *` inside the callback and use as normal.

To avoid needing to process all messages simply switch on the messages you are interested in.

When the event callback is called 'param1' and 'param2' mean different things depending on the type of callback. See `FMOD_EVENT_CALLBACKTYPE` for details.

See Also

- `FMOD_EVENT_CALLBACKTYPE`
- `FMOD_EVENTPROPERTY`
- `Event::setCallback`
Firelight Technologies FMOD Ex
FMOD_MUSIC_CALLBACK

Music callback that is called when one of the events listed in FMOD_MUSIC_CALLBACKTYPE occurs.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_MUSIC_CALLBACK(
    FMOD_MUSIC_CALLBACKTYPE type,
    void * param1,
    void * param2,
    void * userdata
);
```

Parameters

type

Type of callback being issued. see FMOD_MUSIC_CALLBACKTYPE for the different reasons FMOD will generate a callback for the music system.

param1

Parameter 1 for the event callback. See remarks for different uses of param1.

param2

Parameter 2 for the event callback. See remarks for different uses of param2.

userdata

User specified value set when calling MusicSystem::setCallback.

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the
**FMOD_RESULT** enumeration.

**Remarks**

To avoid needing to process all messages simply use a switch statement to select the messages you are interested in.

'param1' and 'param2' mean different things depending on the type of callback. The contents of param1 and param2 are listed below. The parameters are void *, but should be cast to the listed C type to get the correct value.

- **FMOD_MUSIC_CALLBACKTYPE_CHANNEL_CREATED**
  - **param1**: (FMOD_MUSIC_SEGMENT_INFO*) info about the segment that is being played.
  - **param2**: (FMOD::Channel*) the channel that was created.
  This callback is fired when a segment is scheduled to play, just after the channel is created for the segment.

- **FMOD_MUSIC_CALLBACKTYPE_CHANNEL_DESTROYED**
  - **param1**: (FMOD_MUSIC_SEGMENT_INFO*) info about the segment that was being played.
  - **param2**: (FMOD::Channel*) the channel that was destroyed.
  This callback is fired when a segment channel is being destroyed, after the segment has finished playing.

- **FMOD_MUSIC_CALLBACKTYPE_SEGMENT_START**
  - **param1**: (unsigned int) ID of segment being started.
  - **param2**: Unused.

- **FMOD_MUSIC_CALLBACKTYPE_SEGMENT_END**
  - **param1**: (unsigned int) ID of segment being stopped.
  - **param2**: Unused.

- **FMOD_MUSIC_CALLBACKTYPE_SAMPLE_CREATE**
  - **param1**: (FMOD_MUSIC_SAMPLE_INFO*) Info about sample being created.
  - **param2 [out]**: (FMOD::Sound**) pointer to a valid low level FMOD Sound handle. The sound must be created with FMOD_SOFTWARE set. It is also recommended to set **FMOD_2D | FMOD_NONBLOCKING | FMOD_CREATESTREAM**.
- **FMOD_MUSIC_CALLBACKTYPE_SAMPLE_RELEASE**
  - **param1**: `(FMOD_MUSIC_SAMPLE_INFO*)` Info about sample being released.
  - **param2**: `(FMOD::Sound*)` the low level FMOD Sound handle that was previously returned by `FMOD_MUSIC_CALLBACKTYPE_SAMPLE_CREATE`.

- **FMOD_MUSIC_CALLBACKTYPE_RESET**
  - **param1**: Unused.
  - **param2**: Unused.

- **FMOD_MUSIC_CALLBACKTYPE_BEAT**
  - **param1**: (int) bar number (starting at 1).
  - **param2**: (int) beat number (starting at 1).

**See Also**

- **FMOD_MUSIC_CALLBACKTYPE**
- **FMOD_MUSIC_SAMPLE_INFO**
- **FMOD_MUSIC_SEGMENT_INFO**
- **MusicSystem::setCallback**
Firelight Technologies FMOD Ex
Structures

FMOD_EVENT_INFO FMOD_EVENT_LOADINFO
FMOD_EVENT_PROJECTINFO
FMOD_EVENT_SOUNDDEFINFO
FMOD_EVENT_SYSTEMINFO
FMOD_EVENT_WAVEBANKINFO
FMOD_MUSIC_ENTITY
FMOD_MUSIC_INFO
FMOD_MUSIC_ITERATOR
FMOD_MUSIC_SAMPLE_INFO
FMOD_MUSIC_SEGMENT_INFO
Firelight Technologies FMOD Ex
FMOD_EVENT_INFO

Structure containing extended information about an event.

Structure

```c
typedef struct {
    int memoryused;
    int positionms;
    int lengthms;
    int channelsplaying;
    int instancesactive;
    int maxwavebanks;
    FMOD_EVENT_WAVEBANKINFO * wavebankinfo;
    unsigned int projectid;
    unsigned int systemid;
    float audibility;
    int numinstances;
    FMOD_EVENT ** instances;
    FMOD_GUID * guid;
} FMOD_EVENT_INFO;
```

Members

memoryused

This member has been deprecated.

positionms

[out] Time passed in playback of this event instance in milliseconds.

lengthms

[out] Length in milliseconds of this event. Note: lengthms will be -1 if the length of the event can't be determined i.e. if it has looping sounds.

channelsplaying

[out] Number of channels currently playing in this event instance.
instancesactive

[out] Number of event instances currently in use.

maxwavebanks

[in/out] Out, number of wavebanks refered to by this event. In. Maximum size of array of wavebankinfo structures supplied by user. Optional.

wavebankinfo

[in] Pointer to array FMOD_EVENT_WAVEBANKINFO structures (max size defined by maxwavebanks). FMOD will fill these in with detailed information on each wave bank. Optional.

projectid

[out] The runtime 'EventProject' wide unique identifier for this event.

systemid

[out] The runtime 'EventSystem' wide unique identifier for this event. This is calculated when single or multiple projects are loaded.

audibility

[out] current audibility of event.

numinstances

[in/out] On entry, maximum number of entries in instances array. On exit, actual number of entries in instances array, or if instances is null, then it is just the number of instances of this event. Optional.

instances

[in/out] Pointer to an array that will be filled with the current reference-counted event handles of all instances of this event. Optional. Specify 0 if not needed. Must be used in conjunction with numinstances. Note: Due to reference counting, the event instance handles returned here may be different between subsequent calls to this function. If you use these event handles, make sure your
code is prepared for them to be invalid!

\textit{guid}

[out] Pointer to a structure that will be filled with the event's GUID. Optional. Specify 0 if not needed.

\textbf{Remarks}

This structure is optional! Specify 0 or NULL in \texttt{Event::getInfo} if you don't need it!
This structure has members that need to be initialized before \texttt{Event::getInfo} is called. Always initialize this structure before calling \texttt{Event::getInfo}!

\textbf{See Also}

- \texttt{Event::getInfo}
- \texttt{FMOD_GUID}
Firelight Technologies FMOD Ex
FMOD_EVENT_LOADINFO

Use this structure with EventSystem::load when more control is needed over loading.

Structure

typedef struct {
    unsigned int size;
    char * encryptionkey;
    float sounddefentrylimit;
    unsigned int loadfrommemory_length;
    FMOD_BOOL override_category_vals;
    unsigned int sizeof_instancepool_simple;
} FMOD_EVENT_LOADINFO;

Members

size

[in] Size of this structure. This is used so the structure can be expanded in the future and still work on older versions of FMOD Ex.

encryptionkey

[in] Optional. Specify 0 to ignore. Key, or 'password' to decrypt a bank. A sound designer may have encrypted the audio data to protect their sound data from 'rippers'.

sounddefentrylimit

[in] Optional. Specify 0 to ignore. A value between 0 -> 1 that is multiplied with the number of sound definition entries in each sound definition in the project being loaded in order to programmatically reduce the number of sound definition entries used at runtime.

loadfrommemory_length

[in] Optional. Specify 0 to ignore. Length of memory buffer pointed to by
name_or_data parameter passed to `EventSystem::load`. If this field is non-zero then the name_or_data parameter passed to `EventSystem::load` will be interpreted as a pointer to a memory buffer containing the .fev data to load. If this field is zero the name_or_data parameter is interpreted as the filename of the .fev file to load.

`override_category_vals`

[in] Optional. If this member is set to true, newly-loaded categories will impart their properties (volume, pitch etc.) to existing categories of the same name.

`sizeof_instancepool_simple`

[in] Optional. Specify 0 to ignore. If this value is non-zero, FMOD will create an instance pool for simple events with "sizeof_instancepool_simple" entries. Note: Event instance pools currently work for simple events only. Complex events will behave as normal and not be pooled.

**Remarks**

This structure is optional! Specify 0 or NULL in `EventSystem::load` if you don't need it!

Members marked with [in] mean the user sets the value before passing it to the function.
Members marked with [out] mean FMOD sets the value to be used after the function exits.
Use sounddefentrylimit to limit the number of sound definition entries - and therefore the amount of wave data - loaded for each sound definition. This feature allows the programmer to implement a "low detail" setting at runtime without needing a separate "low detail" set of assets.

**See Also**

- `EventSystem::load`
Firelight Technologies FMOD Ex
**FMOD_EVENT_PROJECTINFO**

Structure containing realtime information about an event project.

**Structure**

```c
typedef struct {
    int index;
    char name[256];
    int numevents;
    int numinstances;
    int maxwavebanks;
    FMOD_EVENT_WAVEBANKINFO * wavebankinfo;
    int numplayingevents;
    FMOD_EVENT ** playingevents;
} FMOD_EVENT_PROJECTINFO;
```

**Members**

**index**

[out] Index of the project.

**name**

[out] Name of the project.

**numevents**

[out] Total number of events in all event groups in this event project.

**numinstances**

[out] Total number of event instances in all event groups in this event project.

**maxwavebanks**

wavebankinfo

[in] Pointer to array `FMOD_EVENT_WAVEBANKINFO` structures (max size defined by maxwavebanks). FMOD will fill these in with detailed information on each wave bank. Optional.

numplayingevents

[in/out] On entry, maximum number of entries in playingevents array. On exit, actual number of entries in playingevents array, or if playingevents is null, then it is just the number of currently playing events. Optional.

playingevents

[in/out] Pointer to an array that will be filled with the event handles of all playing events. Optional. Specify 0 if not needed. Must be used in conjunction with numplayingevents.

Remarks

On entry, numplayingevents should be set to the number of elements in the playingevents array. If the actual number of playing events is greater than numplayingevents then the playingevents array will be filled with numplayingevents entries and numplayingevents will be set to the actual number of playing events on exit. In short, if numplayingevents on exit > numplayingevents on entry then the playingevents array wasn't large enough and some events were unable to be added to the array.

See Also

- `EventProject::getInfo`
- `EventSystem::getInfo`
- `FMOD_EVENT_WAVEBANKINFO`
FMOD_EVENT_SOUNDDEFINFO

Structure containing information about a sound definition.

Structure

typedef struct {
    char * name;
    int numentries;
    const char ** entrynames;
    FMOD_EVENT_SOUNDDEF_ENTRYTYPE * entrytypes;
} FMOD_EVENT_SOUNDDEFINFO;

Members

name

The name of the sound definition.

numentries

The number of entries in the sound definition.

entrynames

The names of the entries in the sound definition (an array of size numentries). Note that entrynames[i] will be null if entrytypes[i] is not FMOD_EVENT_SOUNDDEF_ENTRYTYPE_WAVETABLE.

entrytypes

The types of the entries in the sound definition (an array of size numentries).

Remarks

This structure is passed as the param1 argument of all FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_SELECTINDEX callbacks
if FMOD_EVENT_INIT_DETAILED_SOUNDDEF_INFO was passed to
EventSystem::init.

See Also

- **FMOD_EVENT_CALLBACK**
- **FMOD_EVENT_CALLBACKTYPE**
- **EventSystem::init**
- **FMOD_EVENT_INITFLAGS**
- **FMOD_EVENT_SOUNDDEF_ENTRYTYPE**
Firelight Technologies FMOD Ex
FMOD_EVENT_SYSTEMINFO

Structure containing realtime information about an event system.

Structure

typedef struct {
    int numevents;
    int numinstances;
    int maxwavebanks;
    FMOD_EVENT_WAVEBANKINFO * wavebankinfo;
    int numplayingevents;
    FMOD_EVENT ** playingevents;
    int numloadsqueued[FMOD_EVENT_NONBLOCKING_THREAD_MAX];
} FMOD_EVENT_SYSTEMINFO;

Members

numevents

[out] Total number of events in all event groups in this event system.

numinstances

[out] Total number of event instances in all event groups in this event system.

maxwavebanks


wavebankinfo

[in] Pointer to array FMOD_EVENT_WAVEBANKINFO structures (max size defined by maxwavebanks). FMOD will fill these in with detailed information on each wave bank. Optional.

numplayingevents

[in/out] On entry, maximum number of entries in playingevents array. On exit,
actual number of entries in playingevents array, or if playingevents is null, then it is just the number of currently playing events. Optional.

**playingevents**

[in/out] Pointer to an array that will be filled with the event handles of all playing events. Optional. Specify 0 if not needed. Must be used in conjunction with numplayingevents.

**numloadsqueued**

[out] Current number of sound banks queued for loading due to using FMOD_EVENT_NONBLOCKING flag. Note there are multiple possible loading threads depending on what the programmer specified with EventGroup::loadEventData/getEvent/getEventByIndex. Add all queue values together for total.

**Remarks**

On entry, numplayingevents should be set to the number of elements in the playingevents array. If the actual number of playing events is greater than numplayingevents then the playingevents array will be filled with numplayingevents entries and numplayingevents will be set to the actual number of playing events on exit. In short, if numplayingevents on exit > numplayingevents on entry then the playingevents array wasn't large enough and some events were unable to be added to the array.

**See Also**

- EventSystem::getInfo
- EventProject::getInfo
- FMOD_EVENT_WAVEBANKINFO
Firelight Technologies FMOD Ex
**FMOD_EVENT_WAVEBANKINFO**

Structure containing realtime information about a wavebank.

**Structure**

```c
typedef struct {
    char    name[256];
    int     streamrefcnt;
    int     samplerefcnt;
    int     numstreams;
    int     maxstreams;
    int     streamsinuse;
    unsigned int  streammemory;
    unsigned int  samplememory;
    int     type;
} FMOD_EVENT_WAVEBANKINFO;
```

**Members**

- **name**
  
  [out] Name of this wave bank.

- **streamrefcnt**
  
  [out] Number of stream references to this wave bank made by events in this event system.

- **samplerefcnt**
  
  [out] Number of sample references to this wave bank made by events in this event system.

- **numstreams**
  
  [out] Number of times this wave bank has been opened for streaming.

- **maxstreams**
[out] Maximum number of times this wave bank will be opened for streaming.

**streamsInUse**

[out] Number of streams currently in use.

**streamMemory**

[out] Amount of memory (in bytes) used by streams.

**sampleMemory**

[out] Amount of memory (in bytes) used by samples.

**type**

[out] 0 = stream from disk, 1 = load into memory, 2 = decompress into memory.

### See Also

- [EventSystem::getInfo](#)
- [FMOD_EVENT_SYSTEMINFO](#)
Firelight Technologies FMOD Ex
**FMOD_MUSIC_ENTITY**

Structure used to hold information about music system entities.

**Structure**

```c
typedef struct {
    const char * name;
    FMOD_MUSIC_ID id;
} FMOD_MUSIC_ENTITY;
```

**Members**

`* name`

The name of the music entity as a null terminated string.

`id`

The ID of the music entity.

**See Also**

- [FMOD_MUSIC_ITERATOR](#)
Firelight Technologies FMOD Ex
**FMOD_MUSIC_INFO**

Structure containing realtime information about the music system.

**Structure**

```c
typedef struct {
    FMOD_BOOL  starving;
    FMOD_BOOL  all_samples_loaded;
} FMOD_MUSIC_INFO;
```

**Members**

*starving*

[out] True if any streams in the music system are starving.

*all_samples_loaded*

[out] True if all non-streaming samples in the music system are loaded, false otherwise.

**See Also**

- [MusicSystem::getInfo](#)
Firelight Technologies FMOD Ex
**FMOD_MUSIC_ITERATOR**

Structure used to enumerate entities in the music system.

**Structure**

```c
typedef struct {
    const FMOD_MUSIC_ENTITY * value;
    const char * filter;
} FMOD_MUSIC_ITERATOR;
```

**Members**

* `value`

The music entity the iterator points to. A null value indicates an invalid iterator.

* `filter`

The string used to filter music entities.

**Remarks**

The music system provides methods to initialize and advance iterators. Iterator members should never need to be set manually.

**See Also**

- [FMOD_MUSIC_ENTITY](#)
- [MusicSystem::getCues](#)
- [MusicSystem::getParameters](#)
Firelight Technologies FMOD Ex
FMOD_MUSIC_SAMPLE_INFO

Structure containing information about a music sample, for use with
FMOD_MUSIC_CALLBACKTYPE_SEGMENT_CREATE and
FMOD_MUSIC_CALLBACKTYPE_SEGMENT_RELEASE.

Structure

typedef struct {
    unsigned int segment_id;
    unsigned int index;
    const char * filename;
} FMOD_MUSIC_SAMPLE_INFO;

Members

segment_id

The ID of the parent segment.

index

The index of the sample within the parent segment.

filename

The filename of the sample.

Note: If the sample was built by a version of FMOD Designer before 4.29.09,
this field will be 0.

See Also

- FMOD_MUSIC_CALLBACK
- FMOD_MUSIC_CALLBACKTYPE
Firelight Technologies FMOD Ex
FMOD_MUSIC_SEGMENT_INFO

Structure containing information about a music segment, for use with FMOD_MUSIC_CALLBACKTYPE_CHANNEL_CREATED and FMOD_MUSIC_CALLBACKTYPE_CHANNEL_DESTROYED.

Structure

typedef struct {
    unsigned int  segment_id;
    unsigned int  theme_id;
} FMOD_MUSIC_SEGMENT_INFO;

Members

segment_id

The ID of the segment.

theme_id

The ID of the parent theme.

See Also

- [FMOD_MUSIC_CALLBACK](#)
- [FMOD_MUSIC_CALLBACKTYPE](#)
Firelight Technologies FMOD Ex
Defines

EVENT_INITFLAGS EVENT_MEMBITS
EVENT_MODE
EVENT_STATE
Firelight Technologies FMOD Ex
**FMOD_EVENT_INITFLAGS**

Initialization flags. Use them with `EventSystem::init` in the eventflags parameter to change various behaviour.

**Definition**

```c
#define FMOD_EVENT_INIT_NORMAL 0x00000000
#define FMOD_EVENT_INIT_USER_ASSETMANAGER 0x00000001
#define FMOD_EVENT_INIT_FAIL_ON_MAXSTREAMS 0x00000002
#define FMOD_EVENT_INIT_DONTUSENAMES 0x00000004
#define FMOD_EVENT_INIT_UPPERCASE_FILENAMES 0x00000008
#define FMOD_EVENT_INIT_LOWERCASE_FILENAMES 0x00000080
#define FMOD_EVENT_INIT_SEARCH_PLUGINS 0x00000010
#define FMOD_EVENT_INIT_USE_GUIDS 0x00000020
#define FMOD_EVENT_INIT_DETAILED_SOUNDDEF_INFO 0x00000040
#define FMOD_EVENT_INIT_RESETPARAMSTOMINIMUM 0x00000100
#define FMOD_EVENT_INIT_ELEVATION_AFFECTS_LISTENER_ANGLE 0x00000200
#define FMOD_EVENT_INIT_DONTUSELOWMEM 0x00000400
```

**Values**

*FMOD_EVENT_INIT_NORMAL*

All platforms - Initialize normally

*FMOD_EVENT_INIT_USER_ASSETMANAGER*

All platforms - All wave data loading/freeing will be referred back to the programmer through the `FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_CREATE/FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_RELEASE` callback

*FMOD_EVENT_INIT_FAIL_ON_MAXSTREAMS*

All platforms - Events will fail if "Max streams" was reached when playing streamed banks, instead of going virtual.

*FMOD_EVENT_INIT_DONTUSENAMES*
All platforms - All event/eventgroup/eventparameter/eventcategory/eventreverb names will be discarded on load. Use getXXXByIndex to access them. This may potentially save a lot of memory at runtime.

**FMOD_EVENT_INIT_UPPERCASE_FILENAMES**

All platforms - All FSB filenames will be translated to upper case before being used.

**FMOD_EVENT_INIT_LOWERCASE_FILENAMES**

All platforms - All FSB filenames will be translated to lower case before being used.

**FMOD_EVENT_INIT_SEARCH_PLUGINS**

All platforms - Search the current directory for dsp/codec plugins on EventSystem::init.

**FMOD_EVENT_INIT_USE_GUIDS**

All platforms - Build an event GUID table when loading FEVs so that EventSystem::getEventByGUID can be used.

**FMOD_EVENT_INIT_DETAILED_SOUNDDEF_INFO**

All platforms - Pass an FMOD_EVENT_SOUNDDEFINFO struct to FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_SELECTINDEX callbacks rather than just the sound definition name (uses more memory for sound definition waveform names).

**FMOD_EVENT_INIT_RESETPARAMSTOMINIMUM**

All platforms - Reset parameters to minimum value when getting an event instance instead of using the INFO_ONLY event's values.

**FMOD_EVENT_INIT_ELEVATION_AFFECTS_LISTENER_ANGLE**

All platforms - The listener angle event parameters will be affected by elevation, and not just horizontal components.
**FMOD_EVENT_INIT_DONTUSELOWMEM**

All platforms - Instruct the event system to NOT use FMOD_LOWMEM when it opens .FSB files. Specify this flag if you need access to the names of individual subsounds in loaded .FSB files. Specifying this flag will make the event system use more memory.

**See Also**

- `EventSystem::init`
- `EventSystem::getEventByGUID`
- `FMOD_EVENT_SOUNDDINFO`
- `FMOD_EVENT_CALLBACKTYPE`
Firelight Technologies FMOD Ex
FMOD_EVENT_MEMBITS

Bitfield used to request specific memory usage information from the getMemoryInfo function of every public FMOD Event System class. Use with the "event_memorybits" parameter of getMemoryInfo to get information on FMOD Event System memory usage. Every public FMOD Event System class has a getMemoryInfo function which can be used to get detailed information on what memory resources are associated with the object in question. The FMOD_EVENT_MEMBITS defines can be OR'd together to specify precisely what memory usage you'd like to get information on. See EventSystem::getMemoryInfo for an example.

Definition

```c
#define FMOD_EVENT_MEMBITS_EVENTSYSTEM 0x00000001
#define FMOD_EVENT_MEMBITS_MUSICSYSTEM 0x00000002
#define FMOD_EVENT_MEMBITS_FEV 0x00000004
#define FMOD_EVENT_MEMBITS_MEMORYFSB 0x00000008
#define FMOD_EVENT_MEMBITS_EVENTPROJECT 0x00000010
#define FMOD_EVENT_MEMBITS_EVENTGROUPI 0x00000020
#define FMOD_EVENT_MEMBITS_SOUNDBANKCLASS 0x00000040
#define FMOD_EVENT_MEMBITS_SOUNDBANKLIST 0x00000080
#define FMOD_EVENT_MEMBITS_STREAMINSTANCE 0x00000100
#define FMOD_EVENT_MEMBITS_SOUNDDEFCLASS 0x00000200
#define FMOD_EVENT_MEMBITS_SOUNDDEFDEFCLASS 0x00000400
#define FMOD_EVENT_MEMBITS_SOUNDDEFPOOL 0x00000800
#define FMOD_EVENT_MEMBITS_REVERBDEF 0x00001000
#define FMOD_EVENT_MEMBITS_EVENTREVERB 0x00002000
#define FMOD_EVENT_MEMBITS_USERPROPERTY 0x00004000
#define FMOD_EVENT_MEMBITS_EVENTINSTANCE 0x00008000
#define FMOD_EVENT_MEMBITS_EVENTINSTANCE_COMPLEX 0x00010000
#define FMOD_EVENT_MEMBITS_EVENTINSTANCE_SIMPLE 0x00020000
#define FMOD_EVENT_MEMBITS_EVENTINSTANCE_LAYER 0x00040000
#define FMOD_EVENT_MEMBITS_EVENTINSTANCE_SOUND 0x00080000
#define FMOD_EVENT_MEMBITS_EVENTENVELOPE 0x00100000
#define FMOD_EVENT_MEMBITS_EVENTENVELOPEDEF 0x00200000
#define FMOD_EVENT_MEMBITS_EVENTPARAMETER 0x00400000
#define FMOD_EVENT_MEMBITS_EVENTCATEGORY 0x00800000
#define FMOD_EVENT_MEMBITS_EVENTENVELOPEPOINT 0x01000000
#define FMOD_EVENT_MEMBITS_EVENTINSTANCEPOOL 0x02000000
#define FMOD_EVENT_MEMBITS_ALL 0xffffffff
#define FMOD_EVENT_MEMBITS_EVENTINSTANCE_GROUP (FMOD_EVENT_MEMBITS_EVENTINSTANCE || FMOD_EVENT_MEMBITS_EVENTINSTANCE_SIMPLE || FMOD_EVENT_MEMBITS_EVENTINSTANCE_LAYER || FMOD_EVENT_MEMBITS_EVENTINSTANCE_SOUND)
#define FMOD_EVENT_MEMBITS_SOUNDDEF_GROUP (FMOD_EVENT_MEMBITS_SOUNDDEFCLASS || FMOD_EVENT_MEMBITS_SOUNDDEFDEFCLASS || FMOD_EVENT_MEMBITS_SOUNDDEFPOOL)
```
Values

`FMOD_EVENT_MEMBITS_EVENTSYSTEM`
EventSystem and various internals

`FMOD_EVENT_MEMBITS_MUSICSYSTEM`
MusicSystem and various internals

`FMOD_EVENT_MEMBITS_FEV`
Definition of objects contained in all loaded projects e.g. events, groups, categories

`FMOD_EVENT_MEMBITS_MEMORYFSB`
Data loaded with preloadFSB

`FMOD_EVENT_MEMBITS_EVENTPROJECT`
EventProject objects and internals

`FMOD_EVENT_MEMBITS_EVENTGROUP`
EventGroup objects and internals

`FMOD_EVENT_MEMBITS_SOUNDBANKCLASS`
Objects used to manage wave banks

`FMOD_EVENT_MEMBITS_SOUNDBANKLIST`
Data used to manage lists of wave bank usage

`FMOD_EVENT_MEMBITS_STREAMINSTANCE`
Stream objects and internals

`FMOD_EVENT_MEMBITS_SOUNDDEFCLASS`
Sound definition objects

\texttt{FMOD\_EVENT\_MEMBITS\_SOUNDDEFDDEFCLASS}

Sound definition static data objects

\texttt{FMOD\_EVENT\_MEMBITS\_SOUNDDEFPPOOL}

Sound definition pool data

\texttt{FMOD\_EVENT\_MEMBITS\_REVERBDEF}

Reverb definition objects

\texttt{FMOD\_EVENT\_MEMBITS\_EVENTREVERB}

Reverb objects

\texttt{FMOD\_EVENT\_MEMBITS\_USERPROPERTY}

User property objects

\texttt{FMOD\_EVENT\_MEMBITS\_EVENTINSTANCE}

Event instance base objects

\texttt{FMOD\_EVENT\_MEMBITS\_EVENTINSTANCE\_COMPLEX}

Complex event instance objects

\texttt{FMOD\_EVENT\_MEMBITS\_EVENTINSTANCE\_SIMPLE}

Simple event instance objects

\texttt{FMOD\_EVENT\_MEMBITS\_EVENTINSTANCE\_LAYER}

Event layer instance objects

\texttt{FMOD\_EVENT\_MEMBITS\_EVENTINSTANCE\_SOUND}

Event sound instance objects
FMOD_EVENT_MEMBITS_EVENTENVELOPE

Event envelope objects

FMOD_EVENT_MEMBITS_EVENTENVELOPEDEF

Event envelope definition objects

FMOD_EVENT_MEMBITS_EVENTPARAMETER

Event parameter objects

FMOD_EVENT_MEMBITS_EVENTCATEGORY

Event category objects

FMOD_EVENT_MEMBITS_EVENTENVELOPEPOINT

Event envelope point object+s

FMOD_EVENT_MEMBITS_EVENTINSTANCEPOOL

Event instance pool data

FMOD_EVENT_MEMBITS_ALL

All memory used by FMOD Event System

FMOD_EVENT_MEMBITS_EVENTINSTANCE_GROUP

FMOD_EVENT_MEMBITS_SOUNDDEF_GROUP

See Also

- FMOD_MEMBITS
- System::getMemoryInfo
Firelight Technologies FMOD Ex
**FMOD_EVENT_MODE**

Event data loading bitfields. Bitwise OR them together for controlling how event data is loaded. FMOD_EVENT_NONBLOCKING_THREAD0-4. This flag extends FMOD_EVENT_NONBLOCKING to allow multiple asynchronous loads to happen on different threads at the same time.

FMOD_EVENT_NONBLOCKING by itself will always execute on thread 0 by default. Up to 5 simultaneous threads for loading at once are supported.

Only 1 flag should be specified at a time. If multiple flags are specified an FMOD_ERR_INVALID_PARAM error will be returned.

FMOD_EVENT_NONBLOCKING_THREAD0-4 can be specified without the FMOD_EVENT_NONBLOCKING flag being used (it automatically includes it)

**Definition**

```c
#define FMOD_EVENT_DEFAULT 0x00000000
#define FMOD_EVENT_NONBLOCKING 0x00000001
#define FMOD_EVENT_ERROR_ON_DISKACCESS 0x00000002
#define FMOD_EVENT_INFOONLY 0x00000004
#define FMOD_EVENT_USERDSP 0x00000008
#define FMOD_EVENT_NONBLOCKING_THREAD0 (FMOD_EVENT_NONBLOCKING)
#define FMOD_EVENT_NONBLOCKING_THREAD1 (FMOD_EVENT_NONBLOCKING | 0x00010000)
#define FMOD_EVENT_NONBLOCKING_THREAD2 (FMOD_EVENT_NONBLOCKING | 0x00020000)
#define FMOD_EVENT_NONBLOCKING_THREAD3 (FMOD_EVENT_NONBLOCKING | 0x00040000)
#define FMOD_EVENT_NONBLOCKING_THREAD4 (FMOD_EVENT_NONBLOCKING | 0x00080000)
#define FMOD_EVENT_NONBLOCKING_THREAD_MAX 5
```

**Values**

**FMOD_EVENT_DEFAULT**

FMOD_EVENT_DEFAULT specifies default loading behaviour i.e. event data for the whole group is NOT cached and the function that initiated the loading process will block until loading is complete.

**FMOD_EVENT_NONBLOCKING**

For loading event data asynchronously. FMOD will use a thread to load the data. Use Event::getState to find out when loading is complete.
**FMOD_EVENT_ERROR_ON_DISKACCESS**

For `EventGroup::getEvent` / `EventGroup::getEventByIndex`. If `EventGroup::loadEventData` has accidentally been forgotten this flag will return an FMOD_ERR_FILE_UNWANTED if the getEvent function tries to load data.

**FMOD_EVENT_INFOONLY**

For `EventGroup::getEvent` / `EventGroup::getEventByIndex`. Don't allocate instances or load data, just get a handle to allow user to get information from the event.

**FMOD_EVENT_USERDSP**

For `EventGroup::getEvent` / `EventGroup::getEventByIndex`. Tells FMOD that you plan to add your own DSP effects to this event's ChannelGroup at runtime. Omitting this flag will yield a small memory gain.

**FMOD_EVENT_NONBLOCKING_THREAD0**

FMOD_EVENT_NONBLOCKING, execute on thread 0. See remarks. (default)

**FMOD_EVENT_NONBLOCKING_THREAD1**

FMOD_EVENT_NONBLOCKING, execute on thread 1. See remarks.

**FMOD_EVENT_NONBLOCKING_THREAD2**

FMOD_EVENT_NONBLOCKING, execute on thread 2. See remarks.

**FMOD_EVENT_NONBLOCKING_THREAD3**

FMOD_EVENT_NONBLOCKING, execute on thread 3. See remarks.

**FMOD_EVENT_NONBLOCKING_THREAD4**

FMOD_EVENT_NONBLOCKING, execute on thread 4. See remarks.

**FMOD_EVENT_NONBLOCKING_THREAD_MAX**
See Also

- `EventGroup::loadEventData`
- `EventGroup::getEvent`
- `EventGroup::getEventByIndex`
Firelight Technologies FMOD Ex
FMOD_EVENT_STATE

These values describe what state an event is in. The flags below can be combined to set multiple states at once. Use bitwise AND operations to test for these. An example of a combined flag set would be FMOD_EVENT_STATE_READY | FMOD_EVENT_STATE_PLAYING.

Definition

```c
#define FMOD_EVENT_STATE_READY 0x00000001
#define FMOD_EVENT_STATE_LOADING 0x00000002
#define FMOD_EVENT_STATE_ERROR 0x00000004
#define FMOD_EVENT_STATE_PLAYING 0x00000008
#define FMOD_EVENT_STATE_CHANNELSACTIVE 0x00000010
#define FMOD_EVENT_STATE_INFOONLY 0x00000020
#define FMOD_EVENT_STATE_STARVING 0x00000040
#define FMOD_EVENT_STATE_NEEDSTOLOAD 0x00000080
```

Values

**FMOD_EVENT_STATE_READY**

Event is ready to play.

**FMOD_EVENT_STATE_LOADING**

Loading in progress.

**FMOD_EVENT_STATE_ERROR**

Failed to open - file not found, out of memory etc. See return value of Event::getState for what happened.

**FMOD_EVENT_STATE_PLAYING**

Event has been started. This will still be true even if there are no sounds active. Event::stop must be called or the event must stop itself using a 'one shot and stop event' parameter mode.

**FMOD_EVENT_STATE_CHANNELSACTIVE**
Event has active voices. Use this if you want to detect if sounds are playing in the event or not.

*FMOD_EVENT_STATE_INFOONLY*

Event was loaded with the FMOD_EVENT_INFOONLY flag.

*FMOD_EVENT_STATE_STARVING*

Event is streaming but not being fed data in time, so may be stuttering.

*FMOD_EVENT_STATE_NEEDSTOLOAD*

Event still needs to load wavebank data.

**See Also**

- Event::getState
- FMOD_EVENT_MODE
Firelight Technologies FMOD Ex
Enumerations

FMOD_EVENTQUEUE_CALLBACKTYPE
FMOD_EVENT_CALLBACKTYPE
FMOD_EVENT_PITCHUNITS
FMOD_EVENTPROPERTY
FMOD_EVENTRESOURCE
FMOD_EVENT_SOUNDDEF_ENTRYTYPE
FMOD_MUSIC_CALLBACKTYPE
Firelight Technologies FMOD Ex
FMOD_EVENTPROPERTY_TYPE

Event property types.

Enumeration

typedef enum {
    FMOD_EVENTPROPERTY_TYPE_INT,
    FMOD_EVENTPROPERTY_TYPE_FLOAT,
    FMOD_EVENTPROPERTY_TYPE_STRING
} FMOD_EVENTPROPERTY_TYPE;

Values

FMOD_EVENTPROPERTY_TYPE_INT

Property is an int, unsigned int or other type of equivalent size e.g. FMOD_MODE.

FMOD_EVENTPROPERTY_TYPE_FLOAT

Property is a float.

FMOD_EVENTPROPERTY_TYPE_STRING

Property is a char *.

See Also

- Event::getPropertyByIndex
- Event::getPropertyInfo
Firelight Technologies FMOD Ex
These callback types are used with `FMOD_EVENTQUEUE_CALLBACK`.

### Enumeration

typedef enum {
    FMOD_EVENTQUEUE_CALLBACKTYPE_PREPARE,
    FMOD_EVENTQUEUE_CALLBACKTYPE_ABOUTTOPLAY,
    FMOD_EVENTQUEUE_CALLBACKTYPE_FINISHED,
    FMOD_EVENTQUEUE_CALLBACKTYPE_EXPIRED
} FMOD_EVENTQUEUE_CALLBACKTYPE;

### Values

* **FMOD_EVENTQUEUE_CALLBACKTYPE_PREPARE**
  Called when an entry is being prepared for playback

* **FMOD_EVENTQUEUE_CALLBACKTYPE_ABOUTTOPLAY**
  Called when an entry is about to play

* **FMOD_EVENTQUEUE_CALLBACKTYPE_FINISHED**
  Called when an entry has finished playing

* **FMOD_EVENTQUEUE_CALLBACKTYPE_EXPIRED**
  Called when an entry has expired before being played. See `EventQueueEntry::setExpiryTime`

### Remarks

**Note!** Currently the user must call `EventSystem::update` for these callbacks to trigger!
See Also

- EventQueue::setCallback
- FMOD_EVENTQUEUE_CALLBACK
- EventSystem::update
Firelight Technologies FMOD Ex
These callback types are used with `FMOD_EVENT_CALLBACK`.

### Enumeration

```c
typedef enum {
    FMOD_EVENT_CALLBACKTYPE_SYNCPOINT,
    FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_START,
    FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_END,
    FMOD_EVENT_CALLBACKTYPE_STOLEN,
    FMOD_EVENT_CALLBACKTYPE_EVENTFINISHED,
    FMOD_EVENT_CALLBACKTYPE_NET_MODIFIED,
    FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_CREATE,
    FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_RELEASE,
    FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_INFO,
    FMOD_EVENT_CALLBACKTYPE_EVENTSTARTED,
    FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_SELECTINDEX,
    FMOD_EVENT_CALLBACKTYPE_OCCLUSION
} FMOD_EVENT_CALLBACKTYPE;
```

### Values

**`FMOD_EVENT_CALLBACKTYPE_SYNCPOINT`**

Called when a syncpoint is encountered. Can be from wav file markers.

**`FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_START`**

Called when a sound definition inside an event is triggered.

**`FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_END`**

Called when a sound definition inside an event ends or is stopped.

**`FMOD_EVENT_CALLBACKTYPE_STOLEN`**

Called when a getEventXXX call steals an event instance that is in use.

**`FMOD_EVENT_CALLBACKTYPE_EVENTFINISHED`**
Called when an event is stopped for any reason.

```
FMOD_EVENT_CALLBACKTYPE_NET_MODIFIED
```
Called when a property of the event has been modified by a network-connected host.

```
FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_CREATE
```
Called when a programmer sound definition entry is loaded.

```
FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_RELEASE
```
Called when a programmer sound definition entry is unloaded.

```
FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_INFO
```
Called when a sound definition entry is loaded.

```
FMOD_EVENT_CALLBACKTYPE_EVENTSTARTED
```
Called when an event is started.

```
FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_SELECTINDEX
```
Called when a sound definition entry needs to be chosen from a "ProgrammerSelected" sound definition.

```
FMOD_EVENT_CALLBACKTYPE_OCCLUSION
```
Called when an event's channel is occluded with the geometry engine.

**Remarks**

**Note!** Currently the user must call `EventSystem::update` for these callbacks to trigger!

When the event callback is called, 'param1' and 'param2' mean different things depending on the type of callback. Here is an explanation of what these
parameters mean in what context:

**FMOD_EVENT_CALLBACKTYPE_SYNCPOINT**

param1 = (char *) name of sync point  
param2 = (unsigned int) PCM offset of sync point.

An **FMOD_EVENT_CALLBACKTYPE_SYNCPOINT** callback is generated from 'markers' embedded in .wav files. These can be created by placing 'markers' in the original source waves using a tool such as Sound Forge or Cooledit. The waves are then compiled into .FSB files when compiling the audio data using the FMOD designer tool. Callbacks will be automatically generated at the correct place in the timeline when these markers are encountered which makes it useful for synchronization, lip syncing etc.

**FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_START**

param1 = (char *) name of sound definition being started  
param2 = (int) index of wave being started inside sound definition (ie for multi wave sound definitions)

An **FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_START** callback is generated each time a sound definition is played in an event. This happens every time a sound definition starts due to the event parameter entering the region specified in the layer created by the sound designer. This also happens when sounds are randomly respawned using the random respawn feature in the sound definition properties in FMOD designer.

**FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_END**

param1 = (char *) name of sound definition being stopped  
param2 = (int) index of wave being stopped inside sound definition (ie for multi wave sound definitions)
An **FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_END** callback is generated when a one-shot sound definition inside an event ends, or when a looping sound definition stops due to the event parameter leaving the region specified in the layer created by the sound designer.

**FMOD_EVENT_CALLBACKTYPE_STOLEN**

param1 = 0  
param2 = 0  

An **FMOD_EVENT_CALLBACKTYPE_STOLEN** callback is generated when a getEventXXX call needs to steal an event instance that is in use because the event's "Max playbacks" has been exceeded. This callback is called before the event is stolen and before the event is stopped (if it is playing). An **FMOD_EVENT_CALLBACKTYPE_EVENTFINISHED** callback will be generated when the stolen event is stopped i.e. after the **FMOD_EVENT_CALLBACKTYPE_STOLEN**. If the callback function returns **FMOD_ERR_EVENT_FAILED**, the event will not be stolen, and the returned value will be passed back as the return value of the getEventXXX call that triggered the steal attempt.

**FMOD_EVENT_CALLBACKTYPE_EVENTFINISHED**

param1 = 0  
param2 = 0  

An **FMOD_EVENT_CALLBACKTYPE_EVENTFINISHED** callback is generated whenever an event is stopped for any reason including when the user calls Event::stop().

**FMOD_EVENT_CALLBACKTYPE_NET_MODIFIED**

param1 = (**FMOD_EVENT_PROPERTY**) which property was modified  
param2 = (float) the new property value
An **FMOD_EVENT_CALLBACKTYPE_NET_MODIFIED** callback is generated when someone has connected to your running application with FMOD Designer and changed a property within this event, for example volume or pitch.

**FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_CREATE**

param1 = (char *) name of sound definition  
param2 [in] = (int *) pointer to index of sound definition entry  
param2 [out] = (FMOD::Sound **) pointer to a valid lower level API FMOD Sound handle

An **FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_CREATE** callback is generated when a "programmer" sound needs to be loaded.

**FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_RELEASE**

param1 = (char *) name of sound definition  
param2 = (FMOD::Sound *) the FMOD sound handle that was previously created in **FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_CREATE**

An **FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_RELEASE** callback is generated when a "programmer" sound needs to be unloaded.

**FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_INFO**

param1 = (char *) name of sound definition  
param2 = (FMOD::Sound *) the FMOD sound handle that FMOD will use for this sound definition

An **FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_INFO** callback is generated when a sound definition is loaded. It can be used to find information about the specific sound that will be played.
FMOD_EVENT_CALLBACKTYPE_EVENTSTARTED

param1 = 0
param2 = 0

An **FMOD_EVENT_CALLBACKTYPE_EVENTSTARTED** callback is generated whenever an event is started. This callback will be called before any sounds in the event have begun to play.

FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_SELECTINDEX

param1 = (char *) name of sound definition if
*no* `FMOD_EVENT_INIT_DETAILED_SOUNDDEF_INFO` was specified
param1 = (*FMOD_EVENT_SOUNDDEFINFO *) sound definition info struct if
`FMOD_EVENT_INIT_DETAILED_SOUNDDEF_INFO` was specified
param2 [in] = (int *) pointer to number of entries in this sound definition
*param2 [out] = (int) index of sound definition entry to select

An **FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_SELECTINDEX** callback is generated when a sound definition entry needs to be chosen from a "ProgrammerSelected" sound definition.

FMOD_EVENT_CALLBACKTYPE_OCCLUSION

param1 = (float *) pointer to a floating point direct value that can be read and modified after the geometry engine has calculated it for this event's channel.
param2 = (float *) pointer to a floating point reverb value that can be read and modified after the geometry engine has calculated it for this event's channel.

An **FMOD_EVENT_CALLBACKTYPE_OCCLUSION** callback is generated whenever an channel has its occlusion updated via the geometry system.
See Also

- `Event::setCallback`
- `FMOD_EVENT_CALLBACK`
- `FMOD_EVENT_SOUNDEDINFO`
- `FMOD_EVENT_INITFLAGS`
- `EventSystem::update`
Firelight Technologies FMOD Ex
Pitch units for `Event::setPitch` and `EventCategory::setPitch`.

Enumeration

typedef enum {
    FMOD_EVENT_PITCHUNITS_RAW,
    FMOD_EVENT_PITCHUNITS_OCTAVES,
    FMOD_EVENT_PITCHUNITS_SEMITONES,
    FMOD_EVENT_PITCHUNITS_TONES
} FMOD_EVENT_PITCHUNITS;

Values

`FMOD_EVENT_PITCHUNITS_RAW`

Pitch is specified in raw underlying units.

`FMOD_EVENT_PITCHUNITS_OCTAVES`

Pitch is specified in units of octaves.

`FMOD_EVENT_PITCHUNITS_SEMITONES`

Pitch is specified in units of semitones.

`FMOD_EVENT_PITCHUNITS_TONES`

Pitch is specified in units of tones.

See Also

- `Event::setPitch`
- `EventCategory::setPitch`
Firelight Technologies FMOD Ex
FMOD_EVENT_PROPERTY

Property indices for `Event::getPropertyByIndex`.

Enumeration

typedef enum {
    FMOD_EVENTPROPERTY_NAME,
    FMOD_EVENTPROPERTY_VOLUME,
    FMOD_EVENTPROPERTY_VOLUMERANDOMIZATION,
    FMOD_EVENTPROPERTY_PITCH,
    FMOD_EVENTPROPERTY_PITCH_OCTAVES,
    FMOD_EVENTPROPERTY_PITCH_SEMITONES,
    FMOD_EVENTPROPERTY_PITCH_TONES,
    FMOD_EVENTPROPERTY_PITCHRANDOMIZATION,
    FMOD_EVENTPROPERTY_PITCHRANDOMIZATION_OCTAVES,
    FMOD_EVENTPROPERTY_PITCHRANDOMIZATION_SEMITONES,
    FMOD_EVENTPROPERTY_PITCHRANDOMIZATION_TONES,
    FMOD_EVENTPROPERTY_PRIORITY,
    FMOD_EVENTPROPERTY_MAX_PLAYBACKS,
    FMOD_EVENTPROPERTY_MAX_PLAYBACKS_BEHAVIOR,
    FMOD_EVENTPROPERTY_MODE,
    FMOD_EVENTPROPERTY_3D_IGNORE_GEOMETRY,
    FMOD_EVENTPROPERTY_3D_ROLLOFF,
    FMOD_EVENTPROPERTY_3D_MINDISTANCE,
    FMOD_EVENTPROPERTY_3D_MAXDISTANCE,
    FMOD_EVENTPROPERTY_3D_POSITION,
    FMOD_EVENTPROPERTY_3D_CONEINSIDEANGLE,
    FMOD_EVENTPROPERTY_3D_CONEOUTSIDEANGLE,
    FMOD_EVENTPROPERTY_3D_CONEOUTSIDEVOLUME,
    FMOD_EVENTPROPERTY_3D_DOPPLERSCALE,
    FMOD_EVENTPROPERTY_3D_SPEAKERSPREAD,
    FMOD_EVENTPROPERTY_3D_PANLEVEL,
    FMOD_EVENTPROPERTY_SPEAKER_L,
    FMOD_EVENTPROPERTY_SPEAKER_C,
    FMOD_EVENTPROPERTY_SPEAKER_R,
    FMOD_EVENTPROPERTY_SPEAKER_LS,
    FMOD_EVENTPROPERTY_SPEAKER_RS,
    FMOD_EVENTPROPERTY_SPEAKER_LR,
    FMOD_EVENTPROPERTY_SPEAKER_RR,
    FMOD_EVENTPROPERTY_SPEAKER_LFE,
    FMOD_EVENTPROPERTY_REVERBWETLEVEL,
    FMOD_EVENTPROPERTY_ONESHOT,
    FMOD_EVENTPROPERTY_FADEIN,
    FMOD_EVENTPROPERTY_FADEOUT,
}
FMOD_EVENTPROPERTY_REVERBDRYLEVEL,
FMOD_EVENTPROPERTY_TIMEOFFSET,
FMOD_EVENTPROPERTY_SPAWNINTENSITY,
FMOD_EVENTPROPERTY_SPAWNINTENSITYRANDOMIZATION,
FMOD_EVENTPROPERTY_WII_CONTROLLERSPEAKERS,
FMOD_EVENTPROPERTY_3D_POSRANDOMIZATION_MIN,
FMOD_EVENTPROPERTY_3D_POSRANDOMIZATION_MAX,
FMOD_EVENTPROPERTY_EVENTTYPE,
FMOD_EVENTPROPERTY_STEAL_PRIORITY,
FMOD_EVENTPROPERTY_EFFECTS_AFFECT_REVERB,
FMOD_EVENTPROPERTY_WILL_TERMINATE,
FMOD_EVENTPROPERTY_DSPCLOCKSTART_HI,
FMOD_EVENTPROPERTY_DSPCLOCKSTART_LO,
FMOD_EVENTPROPERTY_3D_AUTO_DISTANCE_FILTERING,
FMOD_EVENTPROPERTY_3D_AUTO_DISTANCE_CENTER_FREQ,
FMOD_EVENTPROPERTY_USER_BASE
} FMOD_EVENT_PROPERTY;

Values

**FMOD_EVENTPROPERTY_NAME**

Type: char * - *(ReadOnly)* Name of event.

**FMOD_EVENTPROPERTY_VOLUME**

Type: float - Relative volume of event.

**FMOD_EVENTPROPERTY_VOLUMERANDOMIZATION**

Type: float - Random deviation in volume of event.

**FMOD_EVENTPROPERTY_PITCH**

Type: float - Relative pitch of event in raw underlying units.

**FMOD_EVENTPROPERTY_PITCH_OCTAVES**

Type: float - Relative pitch of event in octaves.

**FMOD_EVENTPROPERTY_PITCH_SEMITONES**

Type: float - Relative pitch of event in semitones.
**FMOD_EVENTPROPERTY_PITCH_TONES**
Type: float - Relative pitch of event in tones.

**FMOD_EVENTPROPERTY_PITCHRANDOMIZATION**
Type: float - Random deviation in pitch of event in raw underlying units.

**FMOD_EVENTPROPERTY_PITCHRANDOMIZATION_OCTAVES**
Type: float - Random deviation in pitch of event in octaves.

**FMOD_EVENTPROPERTY_PITCHRANDOMIZATION_SEMITONES**
Type: float - Random deviation in pitch of event in semitones.

**FMOD_EVENTPROPERTY_PITCHRANDOMIZATION_TONES**
Type: float - Random deviation in pitch of event in tones.

**FMOD_EVENTPROPERTY_PRIORITY**
Type: int - Playback priority of event.

**FMOD_EVENTPROPERTY_MAX_PLAYBACKS**
Type: int - *(Readonly)* Maximum simultaneous playbacks of event.

**FMOD_EVENTPROPERTY_MAX_PLAYBACKS_BEHAVIOR**
Type: int - 1 = steal oldest, 2 = steal newest, 3 = steal quietest, 4 = just fail, 5 = just fail if quietest.

**FMOD_EVENTPROPERTY_MODE**
Type: **FMOD_MODE** - Either **FMOD_3D** or **FMOD_2D**.

**FMOD_EVENTPROPERTY_3D_IGNORE_GEOMETRY**
Type: int - Ignore geometry for that event. 1 = yes, 0 = no.
**FMOD_EVENTPROPERTY_3D_ROLLOFF**

Type: **FMOD_MODE** - Either **FMOD_3D_INVERSEROLLOFF**, **FMOD_3D_LINEARROLLOFF**, **FMOD_3D_LINEARSQUAREROLLOFF**, or none for custom rolloff.

**FMOD_EVENTPROPERTY_3D_MINDISTANCE**

Type: float - Minimum 3d distance of event.

**FMOD_EVENTPROPERTY_3D_MAXDISTANCE**

Type: float - Maximum 3d distance of event. Means different things depending on **EVENTPROPERTY_3D_ROLLOFF**. If event has custom rolloff, setting **FMOD_EVENTPROPERTY_3D_MAXDISTANCE** will scale the range of all distance parameters in this event e.g. set this property to 2.0 to double the range of all distance parameters, set it to 0.5 to halve the range of all distance parameters.

**FMOD_EVENTPROPERTY_3D_POSITION**

Type: **FMOD_MODE** - Either **FMOD_3D_HEADRELATIVE** or **FMOD_3D_WORLDRELATIVE**.

**FMOD_EVENTPROPERTY_3D_CONEINSIDEANGLE**

Type: float - Event cone inside angle. 0 to 360.

**FMOD_EVENTPROPERTY_3D_CONEOUTSIDEANGLE**

Type: float - Event cone outside angle. 0 to 360.

**FMOD_EVENTPROPERTY_3D_CONEOUTSIDEVOLUME**

Type: float - Event cone outside volume. 0 to 1.0.

**FMOD_EVENTPROPERTY_3D_DOPPLERSCALE**

Type: float - Doppler scale where 0 = no doppler, 1.0 = normal doppler, 2.0 = double doppler etc.
FMOD_EVENTPROPERTY_3D_SPEAKERSPREAD
Type: float - Angle of spread for stereo/multichannel source. 0 to 360.

FMOD_EVENTPROPERTY_3D_PANLEVEL
Type: float - 0 = sound pans according to speaker levels, 1 = sound pans according to 3D position.

FMOD_EVENTPROPERTY_SPEAKER_L
Type: float - 2D event volume for front left speaker.

FMOD_EVENTPROPERTY_SPEAKER_C
Type: float - 2D event volume for front center speaker.

FMOD_EVENTPROPERTY_SPEAKER_R
Type: float - 2D event volume for front right speaker.

FMOD_EVENTPROPERTY_SPEAKER_LS
Type: float - 2D event volume for side left speaker.

FMOD_EVENTPROPERTY_SPEAKER_RS
Type: float - 2D event volume for side right speaker.

FMOD_EVENTPROPERTY_SPEAKER_LR
Type: float - 2D event volume for back left speaker.

FMOD_EVENTPROPERTY_SPEAKER_RR
Type: float - 2D event volume for back right speaker.

FMOD_EVENTPROPERTY_SPEAKER_LFE
Type: float - 2D event volume for low frequency speaker.
**FMOD_EVENTPROPERTY_REVERBWETLEVEL**

Type : float - Reverb gain for this event where 0 = full reverb, -60 = no reverb.

**FMOD_EVENTPROPERTY_ONESHOT**

Type : int - Oneshot event - stops when no channels playing. 1 = yes, it's a oneshot 0 = no. it's not a oneshot.

**FMOD_EVENTPROPERTY_FADEIN**

Type : int - Time in milliseconds over which to fade this event in when programmer starts it. 0 = no fade in. Cannot be set while the event is playing.

**FMOD_EVENTPROPERTY_FADEOUT**

Type : int - Time in milliseconds over which to fade this event out when programmer stops it. 0 = no fade out. Cannot be set while the event is playing.

**FMOD_EVENTPROPERTY_REVERBDRYLEVEL**

Type : float - Dry reverb gain for this event where 0 = full dry, -60 = no dry.

**FMOD_EVENTPROPERTY_TIMEOFFSET**

Type : float - Time offset of sound start in seconds.

**FMOD_EVENTPROPERTY_SPAWNINTENSITY**

Type : float - Multiplier for spawn frequency of all sounds in this event.

**FMOD_EVENTPROPERTY_SPAWNINTENSITY_RANDOMIZATION**

Type : float - Random deviation in spawn intensity of event.

**FMOD_EVENTPROPERTY_WII_CONTROLLERSPEAKERS**

Type : int - Wii/WiiU only. Use FMOD_WII_CONTROLLER from fmodwii.h or FMOD_WIIU_CONTROLLER from fmodwiiu.h to set which Wii Controller Speaker(s) to play this event on.
**FMOD_EVENTPROPERTY_3D_POSRANDOMIZATION_MIN**

Type: unsigned int - Minimum radius of random deviation in the 3D position of event.

**FMOD_EVENTPROPERTY_3D_POSRANDOMIZATION_MAX**

Type: unsigned int - Maximum radius of random deviation in the 3D position of event.

**FMOD_EVENTPROPERTY_EVENTTYPE**

Type: int - *(Readonly)* 0 = simple event, 1 = complex event

**FMOD_EVENTPROPERTY_STEAL_PRIORITY**

Type: int - 0 to 10000. How important this event is in relation to other events in the project. This event will never steal an event with a higher steal priority than this.

**FMOD_EVENTPROPERTY_EFFECTS_AFFECT_REVERB**

Type: int - 0 = default (no), 1 = yes. Alternate routing for reverb path of an event so it goes from the layer dsp unit instead of the channel.

**FMOD_EVENTPROPERTY_WILL_TERMINATE**

Type: int - *(Readonly)* 0 = no, 1 = yes, 2 = unknown (current event state is too complex). Whether this event will terminate (stop playing) by itself. If called on an event instance with this_instance = true, the prediction is based on the current state of that instance. This means parameter values, keyoffs etc. come into play.

**FMOD_EVENTPROPERTY_DSPCLOCKSTART_HI**

Type: unsigned int - High 32 bits of a 64 bit DSP clock value, for a start time for any sound in this event.

**FMOD_EVENTPROPERTY_DSPCLOCKSTART_LO**

Type: unsigned int - Low 32 bits of a 64 bit DSP clock value, for a start time for any sound in this event.
**FMOD_EVENTPROPERTY_3D_AUTO_DISTANCE_FILTERING**

Type : int - 0 = default (no), 1 = yes. Whether to automatically apply the distance effect to sounds in this event.

**FMOD_EVENTPROPERTY_3D_AUTO_DISTANCE_CENTER_FREQ**

Type : float - 10 to 22050. The center frequency for the distance effect.

**FMOD_EVENTPROPERTY_USER_BASE**

User created events start from here onwards.

**See Also**

- [Event::getPropertyByIndex](#)
Firelight Technologies FMOD Ex
FMOD_EVENTRESOURCE

Flags to pass to EventGroup::loadEventData to determine what to load at the time of calling.

Enumeration

typedef enum {
    FMOD_EVENTRESOURCE_STREAMS_AND_SAMPLES,
    FMOD_EVENTRESOURCE_STREAMS,
    FMOD_EVENTRESOURCE_SAMPLES
} FMOD_EVENTRESOURCE;

Values

FMOD_EVENTRESOURCE_STREAMS_AND_SAMPLES
Open all streams and load all banks into memory, under this group (recursive)

FMOD_EVENTRESOURCE_STREAMS
Open all streams under this group (recursive). No samples are loaded.

FMOD_EVENTRESOURCE_SAMPLES
Load all banks into memory, under this group (recursive). No streams are opened.

See Also

- EventGroup::loadEventData
Firelight Technologies FMOD Ex
FMOD_EVENT_SOUNDDEF_ENTRYTYPE

Sound definition entry types for FMOD_EVENT_SOUNDDEFINFO.

Enumeration

typedef enum {
    FMOD_EVENT_SOUNDDEF_ENTRYTYPE_WAVETABLE,
    FMOD_EVENT_SOUNDDEF_ENTRYTYPE_OSCILLATOR,
    FMOD_EVENT_SOUNDDEF_ENTRYTYPE_NULL,
    FMOD_EVENT_SOUNDDEF_ENTRYTYPE_PROGRAMMER
} FMOD_EVENT_SOUNDDEF_ENTRYTYPE;

Values

FMOD_EVENT_SOUNDDEF_ENTRYTYPE_WAVETABLE
Waveform.

FMOD_EVENT_SOUNDDEF_ENTRYTYPE_OSCILLATOR
Oscillator.

FMOD_EVENT_SOUNDDEF_ENTRYTYPE_NULL
"Don't play" entry.

FMOD_EVENT_SOUNDDEF_ENTRYTYPE_PROGRAMMER
Programmer sound.

See Also

- FMOD_EVENT_SOUNDDEFINFO
- FMOD_EVENT_CALLBACK
- FMOD_EVENT_CALLBACKTYPE
Firelight Technologies FMOD Ex
FMOD_MUSIC_CALLBACKTYPE

These callback types are used with FMOD_MUSIC_CALLBACK.

Enumeration

typedef enum {
    FMOD_MUSIC_CALLBACKTYPE_SEGMENT_START,
    FMOD_MUSIC_CALLBACKTYPE_SEGMENT_END,
    FMOD_MUSIC_CALLBACKTYPE_SAMPLE_CREATE,
    FMOD_MUSIC_CALLBACKTYPE_SAMPLE_RELEASE,
    FMOD_MUSIC_CALLBACKTYPE_CHANNEL_CREATED,
    FMOD_MUSIC_CALLBACKTYPE_CHANNEL_DESTROYED,
    FMOD_MUSIC_CALLBACKTYPE_RESET,
    FMOD_MUSIC_CALLBACKTYPE_BEAT
} FMOD_MUSIC_CALLBACKTYPE;

Values

FMOD_MUSIC_CALLBACKTYPE_SEGMENT_START

Called when a segment is started.

FMOD_MUSIC_CALLBACKTYPE_SEGMENT_END

Called when a segment ends.

FMOD_MUSIC_CALLBACKTYPE_SAMPLE>Create

Called when a segment needs a sound created.

FMOD_MUSIC_CALLBACKTYPE_SAMPLE_RELEASE

Called when a segment is finished with a sound.

FMOD_MUSIC_CALLBACKTYPE_CHANNEL_CREATED

Called when a channel is created to play a segment.

FMOD_MUSIC_CALLBACKTYPE_CHANNEL_DESTROYED
Called when a segment channel is destroyed.

*FMOD_MUSIC_CALLBACKTYPE_RESET*

Called when the system is reset

*FMOD_MUSIC_CALLBACKTYPE_BEAT*

Called each time a beat is passed (based on segment tempo and time signature)

**Remarks**

**Note!** Currently the user must call `EventSystem::update` for these callbacks to trigger!

**See Also**

- `MusicSystem::setCallback`
- `FMOD_MUSIC_CALLBACK`
- `EventSystem::update`
Firelight Technologies FMOD Ex
C++ Reference

Functions
Firelight Technologies FMOD Ex
Functions

NetEventSystem_GetVersion NetEventSystem_Init
NetEventSystem_Shutdown
NetEventSystem_Update
Firelight Technologies FMOD Ex
NetEventSystem_GetVersion

Get the NetEventSystem version number.

C++ Syntax

```cpp
FMOD_RESULT NetEventSystem_GetVersion(
    unsigned int * version
);
```

C Syntax

```c
FMOD_RESULT FMOD_NetEventSystem_GetVersion(
    unsigned int * version
);
```

Parameters

`version`

A pointer to an integer to receive the version number.

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

See Also

- [NetEventSystem_Init](#)
Firelight Technologies FMOD Ex
NetEventSystem_Init

This function initializes the NetEventSystem and prepares it to accept incoming connections. NOTE: This function must be called before any other NetEventSystem functions.

C++ Syntax

```cpp
FMOD_RESULT NetEventSystem_Init(
    EventSystem * eventsystem,
    unsigned short port
);
```

C Syntax

```c
FMOD_RESULT FMOD_NetEventSystem_Init(
    FMOD_EVENTSYSTEM * eventsystem,
    unsigned short port
);
```

Parameters

**eventsystem**

A pointer to a user-created EventSystem object.

**port**

The TCP port that the NetEventSystem will use to accept incoming connections. 0 = use default port which is 17997.

Return Values

If the function succeeds then the return value is **FMOD_OK**. If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.
Remarks

Specify 0 for the port unless you have a good reason not to. Make sure that whatever port you specify is not blocked.

See Also

- NetEventSystem_Update
- NetEventSystem_Shutdown
- NetEventSystem_GetVersion
Firelight Technologies FMOD Ex
NetEventSystem_Shutdown

Shut down the NetEventSystem.

C++ Syntax

```cpp
FMOD_RESULT NetEventSystem_Shutdown();
```

C Syntax

```c
FMOD_RESULT FMOD_NetEventSystem_Shutdown();
```

Parameters

Return Values

If the function succeeds then the return value is `FMOD_OK`. If the function fails then the return value will be one of the values defined in the `FMOD_RESULT` enumeration.

Remarks

Call this function after you call EventSystem::release.

See Also

- NetEventSystem_Init
Firelight Technologies FMOD Ex
**NetEventSystem_Update**

Update the NetEventSystem.

**C++ Syntax**

```c++
FMOD_RESULT NetEventSystem_Update();
```

**C Syntax**

```c
FMOD_RESULT FMOD_NetEventSystem_Update();
```

**Parameters**

**Return Values**

If the function succeeds then the return value is **FMOD_OK**.
If the function fails then the return value will be one of the values defined in the **FMOD_RESULT** enumeration.

**Remarks**

You must call this function once a frame just after you call EventSystem::update.

**See Also**

- [NetEventSystem_Init](#)
- [NetEventSystem_Shutdown](#)
Firelight Technologies FMOD Ex
PSP::changeOutputMode

This function allows you to switch between FMOD_PSP_OUTPUTMODE_SIMPLEAUDIO and FMOD_PSP_OUTPUTMODE_LIBWAVE on the fly. This allows easier co-existence with movie players. Simple Audio is the most CPU efficient output mode, however it is exclusive so two API's can't use it at once. When playing a movie though another API, you can use this function to switch to libWave while the movie is playing, and then back to Simple Audio again when the movie is finished.

C++ Syntax

```cpp
FMOD_RESULT PSP::changeOutputMode(
    int  libwave_channel
);
```

C Syntax

```c
FMOD_RESULT FMOD_PSP_ChangeOutputMode(
    PSP * psp,
    int  libwave_channel
);
```

Parameters

`libwave_channel`

- Used when switching to libWave. Sets the libWave channel number used for audio output (0 - 2).

Return Values

If the function succeeds then the return value is FMOD_OK.
If the function fails then the return value will be one of the values defined in the FMOD_RESULT enumeration.
FMOD_OK
Remarks

When switching to libWave, it is assumed that the user has taken care of initializing libWave. (via sceWaveInit).