Firelight Technologies FMOD Studio API
Introduction

FMOD Studio is an audio content creation tool for games, with a focus on a Pro Audio approach. It has an interface that will be familiar to those using professional Digital Audio Workstations and is loaded with powerful features.

The programmer API for FMOD Studio comes in 2 flavours. The FMOD Studio API, and the FMOD Studio Low Level API. The FMOD Studio API is the programmer interface to load FMOD Studio banks and trigger events authored by the sound designer. The FMOD Studio Low Level API is the programmer interface that doesn't require any tools, and is just for triggering simple sounds.

For an overview of the Low Level API feature set, read the "Low Level API - An Overview".
What's New

The What's New In 1.10 page highlights the new features of Studio API for this release.
Platform Specific Starter Guide

The Platform Specific Starter Guide has basic platform information to help getting started.
Support

Find out more information at our Questions and Answers database, or contact us directly at support@fmod.com.
Firelight Technologies FMOD Studio API
FMOD Low Level API - An Overview

Table of Contents

What the Low Level API is

API Features

Initialization - Simple start up with no configuration necessary

Audio devices - Automatic detection of device insertion / removal (Windows only)

Audio devices - support for plugins

File formats - Support for over 20 audio formats built in

File formats - Support for the most optimal format for games (FSB)

File formats - Support for plugins

Just play a simple sound - createSound and playSound

High quality / efficient streaming and compressed samples

Streaming

Internet streaming

Streaming settings

Compressed sample playback

Decompressed samples

Voices / Channels - 'Virtual Voices' - play thousands of sounds at once

Voices / Grouping - 'Channel Groups' and hierarchical sub-mixing (buses)
3D sound and spatialization

3D polygon based geometry occlusion

Recording - Record to a sound from microphone or line in

DSP Effects - Support for over 30 special effects built in

DSP Effects - Reverb types and 3D reverb zones

Standard Reverb

Convolution Reverb

Virtual 3D Reverb System

DSP Effects - Support for plugins

DSP Engine - Flexible, programmable soft-synth architecture

Non blocking loads, threads and thread safety

Performance

Configuration - memory and file systems
What the Low Level API is

The FMOD Low Level API is a programmer API that is intended to cover the basics / primitives of sound. This includes concepts such as 'Channels', 'Sounds', 'DSP', 'ChannelGroups', 'Sound Groups', 'Recording' and concepts for 3D Sound and occlusion.

It is standalone and does not require any sound designer tools to interface with. The features are all implemented by the programmer in code.
API Features

This section will give a broad overview of FMOD Low Level API features.

Initialization - Simple start up with no configuration necessary

FMOD Low Level API has an automatic configuration feature, which makes it simple to start.

At the most basic level, creating the System object and calling `System::init` on it. This is all that is needed. A more detailed description of initialization can be found in the FMOD Low Level API Getting Started Tutorial.

The sound card can be manually selected, using the `System::setDriver` function. More settings can be configured, such as the mixing rate of the FMOD system, the resampling method, or the speaker mode with `System::setSoftwareFormat`. When modifying the mixer settings, this only adjusts the internal mixing format. At the end, the audio stream is always converted to the settings that are set by the user (ie the settings in the control panel in Windows, or the standard 7.1/48khz output mode on Xbox One or PS4).

Audio devices - Automatic detection of device insertion / removal (Windows only)

FMOD Low Level API has automatic sound card detection and recovery during playback. If a new device is inserted after initialization, FMOD will seamlessly jump to it, assuming it is the higher priority device. An example of this would be a USB headset being plugged in.

If the device that is being played on is removed (such as a USB audio device), it will automatically jump to the device considered next most important (ie on Windows, it would be the new 'default' device).

If a device is inserted, then removed, it will jump to the device it was originally playing on.

The programmer can override the sound card detection behavior, with a custom
callback. This is the \texttt{FMOD\_SYSTEM\_CALLBACK\_DEVICELISTCHANGED} callback.

\textbf{Audio devices - support for plugins}

FMOD Low Level API has support for user created output plugins. A developer can create a plugin to take FMOD audio output to a custom target. This could be a hardware device, or a non standard file/memory/network based system.

An output mode can run in real-time, or non real-time which allows the developer to run FMOD's mixer/streamer/system at faster or slower than real-time rates.

See \texttt{System::registerOutput} documentation for more.

Plugins can be created inline with the application, or compiled as a stand-alone dynamic library (ie .dll or .so)

\textbf{File formats - Support for over 20 audio formats built in}

FMOD Low Level API has native/built in code to support many file formats out of the box. WAV, MP3 and Ogg Vorbis are supported by default, but many more obscure formats like AIFF, FLAC and others. Sequenced formats that are played back in realtime with a real time sequencer, are included.

MIDI/MOD/S3M/XM/IT are examples of these.

A more comprehensive list can be found in the \texttt{FMOD\_SOUND\_TYPE} list.

\textbf{File formats - Support for the most optimal format for games (FSB)}

FMOD also supports an optimal format for games, called FSB (FMOD Sound Bank).

Many sound file formats are not well suited to games. They are not efficient, and can lead to lots of random file access, large memory overhead, and slow load times.

FSB format benefits are:
• No-seek loading. FSB loading can be 3 continuous file reads. 1. Main header read. 2. Sub-sound metadata. 3. Raw audio data.

• 'Memory point' feature. An FSB can be loaded into memory by the user, and simply 'pointed to' so that FMOD uses the memory where it is, and does not allocate extra memory. See `FMOD_OPENMEMORY_POINT`.

• Low memory overhead. A lot of file formats contain 'fluff' such as tags, and metadata. FSB stores information in compressed, bit packed formats for efficiency.

• Multiple sounds in 1 file. Thousands of sounds can be stored inside 1 file, and selected by the API function `Sound::getSubSound`.

• Efficient Ogg Vorbis. FSB strips out the 'Ogg' and keeps the 'Vorbis'. 1 codebook can be shared between all sounds, saving megabytes of memory (compared to loading .ogg files individually).

• FADPCM codec support. FMOD supports a very efficient, ADPCM variant called FADPCM which is many times faster than a standard ADPCM decoder (no branching), and is therefore very efficient on mobile devices. The quality is also far superior than most ADPCM variants, and lacks the 'hiss' notable in those formats.

**File formats - Support for plugins**

FMOD Low Level API has support for user created file format plugins. A developer can create callbacks for FMOD to call when `System::createSound` or `System::createStream` is executed by the user, or when the decoding engine is asking for data.

Plugins can be created inline with the application, or compiled as a stand-alone dynamic library (ie .dll or .so)

See the `System::registerCodec` documentation for more.

**Just play a simple sound - createSound and playSound**

The simplest way to get started, and basic functionality of FMOD Low Level
API - is to initialize the FMOD system, load a sound, and play it. That's it!

Refer to the [Getting Started tutorial](#) on how to initialize and load/play a sound with the FMOD Low Level API.

Look at the [play sound example](#) to refer to sample code for the simple playback of a sound file.

**High quality / efficient streaming and compressed samples**

FMOD Low Level API benefits from over 15 years of use, in millions of end user devices, causing the evolution of a highly stable and low latency mixing/streaming engine.

**Streaming**

Streaming is the ability to take a large file, and read/play it in realtime in small chunks at a time, avoiding the need to load the entire file into memory. This is typically reserved for

- Music
- Voice over / dialogue
- Long ambience tracks

The user can simply play a sound as a 'stream' by adding the `FMOD_CREATESTREAM` flag to the `System::createSound` function, or using the `System::createStream` function. The 2 options equate to the same end behavior.

**Internet streaming**

FMOD streaming supports internet addresses. Supplying http or https in the filename will switch FMOD to streaming using native http, shoutcast or icecast.

Playlist files (such as ASX/PLS/M3U/WAX formats) are supported, including redirection.

Proxy specification and authentication are supported, as well as real-time
shoutcast stream switching, metadata retrieval and packet loss notification.

**Streaming settings**

Streaming behavior can be adjusted in several ways. As streaming a file takes 2 threads, one for file reading, and one for codec decoding/decompression. File buffer sizes can be adjusted with `System::setStreamBufferSize` and codec decoding buffer size can be adjusted with `FMOD_CREATESOUNDEXINFO` decodeBufferSize member, or `FMOD_ADVANCEDSETTINGS` defaultDecodeBufferSize member.

**Compressed sample playback**

For shorter sounds, rather than decompressing the sound into memory, the user may wish to play the sound in memory, as is.

This is more efficient than a stream, as it does not require disk access, or extra threads to read or decode. A stream has a limit of 1 sound at a time, but a compressed sample does not. It can be played multiple times simultaneously.

If a platform supports a hardware format like AT9 on PS4, or XMA on Xbox One, then it is the best solution to use these codecs, as the decoding of the data is handled by separate media chips, taking the majority of the processing off the CPU.

Refer to the [Getting Started tutorial](#) on how to use the `FMOD_CREATECOMPRESSEDSAMPLE` flag and configuration of codec memory.

**Decompressed samples**

Loading a sound with `System::createSound` will by default, cause a sound to be decompressed into memory, and played back as PCM format.

PCM data is just raw uncompressed sound data, for more information go to the [Terminology / Basic Concepts tutorial](#).

Decompressed / uncompressed samples uses little to no CPU time to process. PCM data is the same format that the FMOD mixing engine uses, and the sound device itself. This may be desirable, if you have enough memory, on a mobile
device with limited CPU cycles.

Decompressed PCM data uses a lot more memory than Vorbis encoded FSB for example. It could be up to 10x more.

A typical use case for mobile developers: Compress the sound heavily for distribution (to reduce the download size), then decompress it at start-up/load time, to save CPU time, rather than playing it compressed.

**Voices / Channels - 'Virtual Voices' - play thousands of sounds at once**

FMOD Low Level includes a 'virtual voice system'. It allows the developer to play hundreds or even thousands of sounds at once, but only have a small number actually producing sound. The remainder are 'virtual' or emulated with a simple position update, and are not heard, and don't consume CPU time.

As an example: a dungeon may have 200 torches burning on the wall in various places but only the loudest torches will be really audible. FMOD will dynamically make voices 'virtual' or 'real' depending on real time audibility calculations (based on distance/volume/priority/occlusion).

A sound which is playing far away, or with a low volume will become virtual, and will change to a real voice when it comes closer, or becomes louder due to Channel or ChannelGroup API calls.

Read more about the virtual voice system on the [Virtual Voice tutorial](#).

**Voices / Grouping - 'Channel Groups' and hierarchical sub-mixing (buses)**

Grouping voices so that they can have a single effect on them, can be achieved with what is typically called a 'bus' or a 'submix'. In FMOD Low Level, channels are grouped into 'Channel Groups', which are the same as a bus or submix.

An effect can be added to a ChannelGroup, and it will only process a sub-mixed result of multiple channels, rather than processing every channel. This reduces
CPU usage greatly.

The volume of a ChannelGroup can be altered, which allows for master volume groups. The volume is scaled based on a fader DSP inside a ChannelGroup. All Channels and Channel Groups have a fader DSP by default.

ChannelGroups are hierarchical. ChannelGroups can contain ChannelGroups, which can contain other ChannelGroups and Channels.

Many attributes can be applied to a ChannelGroup, including things like speaker mix, and 3D position. A whole group of Channels, and the ChannelGroups below them, can be positioned in 3D with 1 call, rather than trying to position all of them individually.

'Master Volume', 'SFX Volume' and 'Music Volume' are typical settings in a game. Setting up an 'SFX' ChannelGroup, and a 'Music' ChannelGroup, and having them children of the master channel group (see System::getMasterChannelGroup)

3D sound and spatialization

FMOD Low Level API has support for a variety of features that allow sounds to be placed in 3D space, so that they move around the listener as part of an environment, by panning, pitch shifting with doppler, and attenuating with volume scaling or even special filtering.

FMOD 3D spatialization features:

1. Multiple attenuation rolloff models. Rolloff is the behavior of the volume of the sound as the sound gets closer to the listener or further away. Choose between linear, inverse, linear square, inverse tapered and custom rolloff modes. Custom rolloff allows a FMOD_3D_ROLLOFF_CALLBACK to be set to allow the user to calculate how the volume rolloff happens. If a callback is not convenient, FMOD also allows an array of points that are linearly interpolated between, to denote a 'curve', using ChannelControl::set3DCustomRolloff.

2. Doppler pitch shifting. Accurate pitch shifting, controlled by the user velocity setting of the listener and the channel or channelgroup, is
calculated and set on the fly by the FMOD 3D spatialization system.

3. Vector Based Amplitude Panning (VBAP). This system pans the sounds in the user's speakers in real time, supporting mono, stereo, up to 5.1 and 7.1 surround speaker setups.

4. Occlusion. Channels or ChannelGroups can have lowpass filtering applied to them to simulate sounds going through walls or being muffled by large objects.

5. 3D Reverb Zones for reverb panning. See more about this in the 3D Reverb section. Reverb can also be occluded to not go through walls or objects.

6. Polygon based geometry occlusion. Add polygon data to FMOD's geometry engine, and FMOD will automatically occlude sounds in realtime using raycasting. See more about this in the 3D Polygon based geometry section.

7. Multiple listeners. In a split screen mode game, FMOD can support a listener for each player, so that the 3D sounds attenuate correctly.

8. Morphing between 2D and 3D with multichannel sounds. Sounds can be a point source, or be morphed by the user into a 2D sound, which is great for distance based envelopment. The closer a sound is, the more it can spread into the other speakers, rather than flipping from one side to the other as it pans from one side to the other. See Channel::set3DLevel or ChannelGroup::set3DLevel for the function that lets the user change this mix.

9. Stereo and multichannel sounds can be 3D. Typically a mono sound is used for 3D audio. Multi-channel sounds can be used to give extra impact. By default a multichannel sound is collapsed into a mono point source. To 'spread' the channels of the multichannel sound, use Channel::set3DSpread or ChannelGroup::set3DSpread. This can give a more a more spatial effect for a sound that is coming from a certain direction. A subtle spread of sound in the distance may gives the impression of being more effectively spatialized as if it were reflecting off nearby surfaces, or being 'big' and emitting different parts of the sound in different directions.

10. Spatialization plugin support. 3rd party VR audio plugins can be used to give more realistic panning over headphones.
To load a sound as 3D simply add the FMOD_3D flag to the System::createSound function, or the System::createStream function.

The next 3 important things to do are:

1. Set the 'listener' position, orientation and velocity once per frame with System::set3DListenerAttributes.

2. Set the Channel 3D attributes for handle that was returned from System::playSound, with Channel::set3DAttributes. If 3D positioning of a group of channels, or a ChannelGroup is required, set the ChannelGroup to be 3D once with ChannelGroup::setMode, then call ChannelGroup::set3DAttributes instead.

3. Call System::update once per frame so the 3D calculations can update based on the positions and other attributes.

Read more about 3D sound in the 3D Sound tutorial or the Spatial Audio tutorial.

3D polygon based geometry occlusion

FMOD Low Level API supports the supply of polygon mesh data, that can be processed in realtime to create the effect of occlusion in a real 3D world. In real world terms, the user can stop sounds travelling through walls, or even confine reverb inside a geometric volume so that it doesn't leak out into other areas.

Geometry can be used to obstruct, occlude and exclude dry/reverb signals (yellow = direct, green = reverb)

To use the FMOD Geometry Engine, create a mesh object with System::createGeometry. Then add polygons to each mesh with Geometry::addPolygon. Each object can be translated, rotated and scaled to fit your environment.

Recording - Record to a sound from microphone or line in
FMOD Low Level API has the ability to record directly from an input into an FMOD sound object.

This sound can then be played back after it has been recorded, or the raw data can be retrieved with `Sound::lock` and `Sound::unlock` functions.

The sound can also be played while it is recording, to allow realtime effects. A simple technique to achieve this is to start recording, then wait a small amount of time, like 50ms, then play the sound. This will keep the play cursor just behind the record cursor. See the record example for source and information on how to do this.

**DSP Effects - Support for over 30 special effects built in**

FMOD Low Level API has native/built in code to support many special effects out of the box, such as low-pass, compressor, reverb and parametric EQ. A more comprehensive list can be found in the `FMOD_DSP_TYPE` list.

An effect can be created with `System::createDSPByType` and added to a Channel with `Channel::addDSP` or a ChannelGroup with `ChannelGroup::addDSP`.

**DSP Effects - Reverb types and 3D reverb zones**

FMOD Low Level API has 2 types of physical reverb available, and a virtual 3d reverb system which can be used to simulate hundreds of environments or more, with only 1 physical reverb.

**Standard Reverb**

A built in, high quality I3DL2 standard compliant reverb, which is used for a fast, configurable environment simulation, and is used for the 3D reverb zone system, described below.

To set an environment simply, use `System::setReverbProperties`. This lets you set a global environment, or up to 4 different environments, which all sounds are affected by.

Each channel can have a different reverb wet mix by setting the level in `Channel::setReverbProperties`. 
Read more about the I3DL2 configuration in the Reverb Notes section of the documentation. To avoid confusion when starting out, simply play with the preset list of environments in `FMOD_REVERB_PRESETS`.

**Convolution Reverb**

There is also an even higher quality Convolution Reverb which allows a user to import an impulse response file (a recording of a impulse in an environment which is used to convolve the signal playing at the time), and have the environment sound like it is in the space the impulse was recorded in.

This is an expensive to process effect, so FMOD supports GPU acceleration to offload the processing to the graphics card. This greatly reduces the overhead of the effect to being almost negligible. GPU acceleration is supported on Xbox One and PS4 platforms.

Convolution reverb can be created with `System::createDSPByType` with `FMOD_DSP_TYPE_CONVOLUTIONREVERB` and added to a ChannelGroup with `ChannelGroup::addDSP`. It is recommended to only implement 1 or a limited number of these effects and place them on a sub-mix/group bus (a ChannelGroup), and not per channel.

**Virtual 3D Reverb System**

A Virtual 3D reverb zone system is supported, using the main built-in system I3DL2 reverb.

Virtual '3D reverb spheres' can be created and placed around a 3D world, in unlimited numbers, causing no extra CPU expense.

As the listener travels through these spheres, FMOD will automatically morph and attenuate the levels of the system reverb to make it sound like you are in different environments as you move around the world.

Spheres can be overlapped and based on where the listener is within each spheres. FMOD will morph the reverb to the appropriate mix of environments.
A 3D reverb sphere can be created with `System::createReverb3D` and the position set with `Reverb3D::set3DAttributes`. To set a sphere's reverb properties, `Reverb3D::setProperties` can be used.

For more information on the 3D reverb zone system, and implementation information, read the [3D Reverb Tutorial](#).

**DSP Effects - Support for plugins**

FMOD Low Level API has support for user created DSP plugins. A developer can either load a pre-existing plugin, or create one inside the application, using 'callbacks'.

Callbacks can be specified by the user, for example when `System::createDSP` is called, or when the DSP runs and wants to process PCM data inside FMOD's mixer.
Plugins can be developed inline with the application, or compiled as a stand-alone dynamic library (ie .dll or .so)

To load a pre-existing plugin executable, use the System::loadPlugin function.

To implement callbacks directly in a program, System::registerDSP can be used.

To create a stand alone dynamic library, use the same callbacks, but export the symbols through a the FMOD_DSP_DESCRIPTION struct, via the exported FMODGetDSPDescription function.

See the DSP Plugin API tutorial on how to make a plugin, and /examples/fmod_gain.cpp in the API distribution as a working example.

**DSP Engine - Flexible, programmable soft-synth architecture**

FMOD Low Level API runs on a modular synth architecture, which allows connections of signal processing nodes (the 'FMOD DSP' concept. See the FMOD DSP Class to be joined together to create deeper more complicated audio signals and flow.

A directed graph processing tree allows the signal to flow from 'generators' (a sound playing through from System::playSound, or a DSP creating sound from System::playDSP for example), to other nodes, mixing together until they reach the head node, where the final result is sent to the sound card.
A visual representation taken directly from the FMOD Profiler tool (in the /bin directory of the API distribution).

FMOD typically processes the sound in the graph, in blocks of 512 samples (10ms) on some platforms, or 1024 on other platforms (21ms). This is the granularity of the system, and affects how smooth parameter changes, such as pitch or volume will heard.

FMOD pre-built DSP effects can be inserted into the graph with functions like `DSP::addInput` and `DSP::disconnectFrom`.

For detailed information read the DSP Architecture and Usage tutorial.

**Non blocking loads, threads and thread safety**

FMOD Low Level API commands are thread safe and queued. They get processed either immediately, or in background threads, depending on the command.

By default, things like initialization, and loading a sound are processed on the main thread.
Mixing, streaming, geometry processing, file reading and file loading are or can be done in the background, in background threads. Every effort is made to avoid blocking the main application's loop unexpectedly.

One of the slowest operations is loading a sound. To place a sound load into the background so that it doesn't affect processing in the main application thread, the user can use the `FMOD_NONBLOCKING` flag in `System::createSound` or `System::createStream`.

Thread affinity is configurable on some platforms.

For detailed information about FMOD and threads please refer to the Threads and Thread Safety tutorial.
Performance

The FMOD Low Level API has evolved over the years to have a comprehensive suite of effects and codecs with minimal overhead for memory and CPU.

All platforms come with performance saving features. For example vector optimized floating point math is used heavily. Some of the technologies used include SSE, NEON, AVX, VMX, and VFP assembler.

Typically the most expensive part of sound playback is real-time compressed sample playback.

The FMOD API will allow configuration of how many sounds should be audible at once, to reduce CPU overhead. This is configurable as mentioned in the Compressed sample playback section of this document, using the System::setAdvancedSettings function.

Adjusting the sample rate quality, resampling quality, number of mixed voices and decoded voices is configurable to get the best scalability for your application.

To find out more about configuring FMOD to save CPU time, refer to the CPU Performance tutorial, or to get an idea about Low Level performance figures on various platforms, refer to the Performance Reference section of the documentation.
Configuration - memory and file systems

The FMOD Low Level API caters to the needs of applications and their memory and file systems. A file system can be 'plugged in' so that FMOD uses it, and not its own system, as well as memory allocation.

To set up a custom file system is a simple process of calling System::setFileSystem.

The file system handles the normal cases of open, read, seek, close, but adds an extra feature which is useful for prioritized/delayed file systems, FMOD supports the FMOD_FILEASYNCREAD_CALLBACK callback, for deferred, prioritized loading and reading, which is a common feature in advanced game streaming engines.

An async read callback can immediately return without supplying data, then when the application supplies data at a later time, even in a different thread, it can set the 'done' flag in the FMOD_ASYNCREADINFO structure to get FMOD to consume it. Consideration has to be made to not wait too long or increase stream buffer sizes, so that streams don't audibly stutter/skip.

To set up a custom memory allocator is done by calling FMOD_Memory_Initialize. This is not an FMOD class member function because it needs to be called before any FMOD objects are created, including the System object.

To read more about setting up memory pools or memory environments, refer to the Memory Management tutorial.
Firelight Technologies FMOD Studio API
What's New in 1.10

This section describes the major features introduced in the 1.10 release. See the Detailed Revision History for information regarding each patch release.
**Spatial audio features**

Windows Sonic spatialization has been added for Windows and Xbox One with a new output plugin **FMOD_OUTPUTTYPE_WINSONIC**. This will allow FMOD to be rendered using Windows Sonic for headphones, Dolby Atmos for headphones and Dolby Atmos Home Theatre. These technologies allow for a more immersive surround experience which includes height spatialization via 7.1.4 surround speaker mode and dynamic objects.

To facilitate getting signal into the height speakers FMOD can play 12 channel audio (7.1.4) as a source or upmix with the help of **FMOD_DSP_TYPE_PAN** and the new **FMOD_DSP_PAN_2D_HEIGHT_BLEND** parameter.

For more detail about using spatial audio features with FMOD please refer to the dedicated [Spatial Audio](#) page.
Firelight Technologies FMOD Studio API
Previous Releases

- What's New In 1.08
- What's New In 1.07
- What's New In 1.06
- What's New In 1.05
- What's New In 1.04
- What's New In 1.03
- What's New In 1.02
- What's New In 1.01
Firelight Technologies FMOD Studio API
What's New in 1.09

This section describes the major features introduced in the 1.09 release. See the Detailed Revision History for information regarding each patch release.
Multiple listener weighting

Studio's multiple listener support has been improved with the ability to set a weighting per listener with Studio::System::setListenerWeight. The weight allows listeners to be smoothly faded in or out of existence.

For example to do a crossfade, an additional listener can be created with a weighting of 0 that ramps up to 1 while the old listener weight is ramped down to 0. After the crossfade is finished the number of listeners can be set down to 1 again.

Listener weight is used in the 3D panner, the doppler calculation, and the automatic distance event parameter.

For more information, see the Studio 3D Events page.
Getting final value for volume and parameter values

The Studio API getter functions take an extra optional argument which will receive the final value after automation, modulation, and snapshots.

It allows the sound designer to set up automations on parameters which the game can query and drive other game side systems, as well as getting volume of buses after snapshots are applied. For existing codebases, the second argument defaults to NULL which is unchanged behaviour.

The functions that have the extra functionality are:

- `Studio::EventInstance::getVolume`
- `Studio::EventInstance::getPitch`
- `Studio::EventInstance::getParameterValue`
- `Studio::EventInstance::getParameterValueByIndex`
- `Studio::Bus::getVolume`
- `Studio::VCA::getVolume`

As part of this change, the older function names have been renamed for consistency:

- `Studio::Bus::setFaderLevel` is now `Studio::Bus::setVolume`
- `Studio::Bus::getFaderLevel` is now `Studio::Bus::getVolume`
- `Studio::VCA::setFaderLevel` is now `Studio::VCA::setVolume`
- `Studio::VCA::getFaderLevel` is now `Studio::VCA::getVolume`
Multiband EQ

A new EQ DSP has been developed to roll up several simpler DSPs into one high performance flexible effect.

The multiband EQ consists of 5 independently configurable parametric equalizers with several filtering modes:

- Low-pass (12dB, 24dB, 48dB)
- High-pass (12dB, 24dB, 48dB)
- Low-shelf
- High-shelf
- Peaking
- Band-pass
- Notch
- All-pass

This new effect will supercede existing lowpass, highpass, lowpass simple, highpass simple and parametric eq effects. It has better performance and a superset of functionality compared with the now deprecated effects and thus should be considered as a replacement. For more information on how to use the new effect as a replacement see FMOD_DSP_TYPE remarks.
What's new since 1.08 initial release

This section covers some of the improvements introduced between the first 1.08 release and the new 1.09 release. See the Detailed Revision History for more information on features introduced during the 1.08 lifetime.
Sound Play / Stop callbacks

You can now set a callback to be fired when the Studio runtime plays or stops an events. With the callback is the low level FMOD::Sound through which you can query the name of the sound played
Listener mask

Events can be set up to only be influenced by some of the listeners. `Studio::EventInstance::setListenerMask` can be used to specify a mask for what listeners apply to that event.

For more information, see the Studio 3D Events page.
Firelight Technologies FMOD Studio API
What's New in 1.08

This section describes the major features introduced in the 1.08 release. See the Detailed Revision History for information regarding each patch release.
Bank loading performance improvements

Bank sample data loading has been optimised. The number of file accesses and seeks has been reduced. Multiple sample data loads are coalesced into a single read operation in sorted order. Performance when cancelling pending sample data loads has been improved. There have been memory savings, so projects that load large numbers of sounds will see a memory improvement.

Studio sample data loading occurs on its own thread. Affinity can be controlled via per platform affinity settings.
Sample Data Idle Pool

To avoid spurious sample data reloading, Studio will keep a pool of recently used sounds in memory. This can help for cases where programmer doesn't manually preload data via `Studio::EventDescription::loadSampleData` or `Studio::Bank::loadSampleData`. See the Studio Bank Programmer Topic for more information.
Studio Profiler file I/O display

The FMOD Studio Profiler now displays the file I/O used for sample data, stream data, and bank metadata loading. The event instance lifetime view has indicators for when event instances are stalling waiting for sample data to load in.
3D Object Panner

The new 3D object panner DSP (FMOD_DSP_TYPE_OBJECTPAN) allows channels to be spatialized using Dolby Atmos (FMOD_OUTPUTTYPE_ATMOS) and Playstation VR (FMOD_OUTPUTTYPE_AUDIO3D) technology.

The output mode plugin API (FMOD_OUTPUT_DESCRIPTION) has been extended to allow custom object panner backends.

There is a new example 'object_pan' that demonstrates object based panning.
Multiple plugins in one file

Multiple DSP, output and codec plugins can be packaged into a single module. See `FMOD_PLUGINLIST`, `System::getNumNestedPlugins`, `System::getNestedPlugin`, and the DSP Plugin API Programmer Topic for more information.
Sound loading performance

Increased performance when loading samples and opening streams via the low level API.
Low latency output mode for Windows

The default WASAPI output mode has reduced latency by 40ms and improved mixer thread regularity.
**FSBank and Profiler tools for Mac**

Added Mac versions of the Profiler tool, FSBank tool, and FSBankLib API. These are included in the Mac and iOS API packages.
Support for new Studio features

- Left edge trimming of timelocked sounds.
- Start Offset as a percentage of sound length.
- Per asset encoding and streaming settings.
- New 'Decompressed' loading mode allows sounds to be decompressed into memory when loaded, rather than staying compressed.
What's new since 1.07 initial release

This section covers some of the improvements introduced between the first 1.07 release and the new 1.08 release. See the Detailed Revision History for more information on features introduced during the 1.07 lifetime.
**Transceiver effect**

This new DSP effect (**FMOD_DSP_TYPE_TRANSCEIVER**) broadcasts a signal to one of 32 global 'slots' or 'channels' (like a radio station). These signals can be monitored by a receiver anywhere in the mix. Each channel may have multiple transmitters or receivers connected.

Applications include sending the output of a global reverb back out into the world to be spacialized, playing in-game radio both inside the cabin and also through loudspeakers placed outside, and many more.
Firelight Technologies FMOD Studio API
What's New in 1.07

This section describes the major features introduced in the 1.07 release. See the Detailed Revision History for information regarding each patch release.
64-bit version of Studio

Studio is now a 64-bit application, providing greater memory capacity and improved performance. On Windows, separate 32-bit and 64-bit versions are available. On Mac, Studio is now 64-bit only.

All plugin libraries must be built for 64-bit architecture in order to be compatible with the 64-bit version of Studio.
Nested events have been converted to reference events

The concept of nested events and referenced events has been unified to just having reference events. As part of this change, Studio::Bank::getEventCount and Studio::Bank::getEventList now only return top level events that have been explicitly added to banks by the sound designer. They do not include any reference events that have been included in the bank implicitly via event instrument references.
**Bus Polyphony**

This setting allows the designer to set a limit on the number of event instances that route through a bus.
Improved recording and enumeration support

When using `System::getRecordNumDrivers` and `System::getRecordDriverInfo` you can now be sure that a particular list index will always be valid. As devices are added the list will grow over time, if a device is removed it will remain in the list. You can now query `FMOD_DRIVER_STATE` to determine `FMOD_DRIVER_STATE_CONNECTED`.

To take advantage of these improvements all major platforms support dynamic device addition and removal allowing developers to query the current state of all recording hardware. For notifications about changes `System::setCallback` can be used with `FMOD_SYSTEM_CALLBACK_RECORDLISTCHANGED`. All recording device changes are now decoupled from output changes.
Multi-channel compressed sample support

In previous versions of FMOD multi-channel (greater than stereo) sounds could be streamed or played after being decompressed into memory, but not as an in memory compressed sound (ie Vorbis). This release allows any FSB compressed sound to be multi-channel, accessible via \texttt{FMOD\_CREATECOMPRESSEDSAMPLE} flag if using the low level API, or just drag and drop multi-channel sounds into FMOD Studio's UI and play as normal.
Added **FMOD_DSP_TYPE_CHANNELMIX** effect

This new DSP effect allows the user to control gain levels for up to 32 input channel signals, and pipe the output to a range of speaker formats i.e. repeating mono, stereo, quad, 5.1, 7.1 or only LFE out.
Performance improvements

- Improved Vorbis decoding performance on Windows.
- Metering / Profiling performance significantly improved.
Platform specific improvements

- New platforms - Universal Windows Platform (UWP) and AppleTV support.
- Mac - Added support for recording from multiple microphones at the same time.
- iOS - Added support for bitcode.
- Win - Reduced dll and exe sizes.
New Plugins

- AudioGaming - AudioMotors. Create realtime interactive car engine sounds from a single recorded engine sweep!
- Two Big Ears - 3DCEPTION. 3Dception is a real-time 3D audio and environmental modelling engine.
What's new since 1.06 initial release

This section covers some of the improvements introduced between the first 1.06 release and the new 1.07 release.

- Internet streams. Add Sound::setPosition and `Sound::seekData` support for HTTP streams.
- UE4 - Added blueprint functions for loading and unloading sample data, new initialization settings, beat and marker callbacks as Blueprint events, reverb zones, and 4.9 engine preview build.
- Studio Profiler now shows nested instance information.
- Reduced memory overhead of bank metadata.
- `FMOD_STUDIO_LOAD_BANK_DECOMPRESS_SAMPLES` flag has been added to allow compressed banks to be decompressed at load time, to help low spec machines.
- Tempo and marker event callbacks added to Studio API. See `FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_MARKER` and `FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_BEAT`.
- Additional event callback types: `FMOD_STUDIO_EVENT_CALLBACK_CREATED`, `FMOD_STUDIO_EVENT_CALLBACK_DESTRUCTED` and `FMOD_STUDIO_EVENT_CALLBACK_START_FAILED`.
- `FMOD_DSP_COMPRESSOR` now supports unlinked channel compression, so multichannel sidechaining for example can compress each destination channel individually.
- DSP Plugin writers - `FMOD_DSP_DESCRIPTION::sys_register`, `sys_deregister` and `sys_mix` callbacks have been added, to allow a per type (not instance) level init/shutdown/mix callback for a plugin.

See the [Detailed Revision History](#) for more information on features introduced during the 1.06 lifetime.
Firelight Technologies FMOD Studio API
What's New in 1.06

This section describes the major features introduced in the 1.06 release. See the Detailed Revision History for information regarding each patch release.
Bank Loading Changes

Studio now does all file loading on a separate loading thread. This avoids any hitches in scheduling that may occur when loading large banks. The interface is unchanged since the `FMOD_STUDIO_LOAD_BANK_NONBLOCKING` flag already exists.

Banks loaded with `FMOD_STUDIO_LOAD_BANK_NONBLOCKING` will no longer be destroyed if the loading fails, instead they will remain in an error state that can be queried with `Studio::Bank::getLoadingState`. Banks that enter the error state should be released by calling `Studio::Bank::unload`.

Several enhancements were made for `FMOD_STUDIO_LOAD_MEMORY_POINT`. Extra data can be associated with the bank by `Studio::Bank::getUserData` and `Studio::Bank::setUserData`. There is a new callback `FMOD_STUDIO_SYSTEM_CALLBACK_BANK_UNLOAD` which is called when banks are asynchronously unloaded. That provides a convenient place to free any memory associated with the bank.

The load_banks example has been updated to demonstrate all the methods of loading banks.
API Capture and Replay

API command capture functionality has been improved. API commands are now captured when recording a profiling session in Studio, allowing sound designers to create API command captures without the need for programmer support.

API captures now write out the initial state of the playback system including currently loaded banks as well as the state of all created event instances. This allows API captures to be taken mid-game without missing any information.

Studio::System::loadCommandReplay now returns a CommandReplay object that has functionality for querying information as well as starting, stopping, and pausing playback. Playback is now asynchronous.
Multiple listener support

The Studio API has been updated to support multiple listeners. Studio::System::setNumListeners and Studio::System::getNumListeners has been added, and the existing Studio::System::setListenerAttributes and Studio::System::getListenerAttributes functions now take a listener index.

Doppler is disabled when in multiple listener mode.

As part of this change, there is a new DSP data parameter FMOD_DSP_PARAMETER_3DATTRIBUTES_MULTI that can be used for DSP panners that want to support multiple listener panning.
FADPCM Compression

This new compression format is recommended for all developers targeting mobile devices, PS Vita or Wii U and is considered a drop in replacement for IMA ADPCM compression. It achieves a slightly higher compression ratio, 3.66:1 compared with 3.55:1 while being a step up in quality. However the main benefit of this new custom format is the ground up design for efficient decoding, in some cases twice as fast as IMA ADPCM.
Firelight Technologies FMOD Studio API
What's New in 1.05

This section describes the major features introduced in the 1.05 release. See the Detailed Revision History for information regarding each patch release.
MixerStrip Replaced by Bus and VCA

The Studio::MixerStrip class has been replaced by Studio::Bus and Studio::VCA to better reflect the differences between the two concepts. The new classes do not have a release method, as their handles remain valid until the bank containing them is unloaded.
Get Event functions

The Studio::System::getEvent, Studio::System::getBus, Studio::System::getVCA and Studio::System::getBank functions now take a string that can either be the full path or the string representation of the ID. This avoids the additional call to Studio::parseID or Studio::System::lookupID in most cases.

The functions to get by ID are now named Studio::System::getEventByID, Studio::System::getBusByID, Studio::System::getVCAByID, and Studio::System::getBankByID.
Bus ChannelGroup Lifetime

Each bus's channelgroup is now created and destroyed on demand. This means it only exists if at least one event instance routes into the bus. If it doesn't exist, Studio::Bus::getChannelGroup will return FMOD_ERR_STUDIO_NOT_LOADED.
**Event Instance Release Behavior Change**

`Studio::EventInstance::release` no longer invalidates the handle immediately. Instead, the handle remains valid until the instance stops and is actually destroyed. In addition, this function no longer checks if the event will stop naturally. Looping events will not be destroyed until they are manually stopped.

This change also means that event callbacks will continue to be fired after `Studio::EventInstance::release` is called until the instance is actually destroyed. This can be prevented by setting the callback to NULL before calling `Studio::EventInstance::release`. 
Event Stop Behavior Change

Events with sounds triggered by parameters will now stop once all sounds have finished and the timeline cursor has reached the end. Previously these events would go idle instead of stopping. The old behavior can be achieved by adding a sustain point to the timeline.
Deprecated Function Removed

Studio::EventInstance::createSubEvent has been removed. Use Event Sounds instead.
**Wide String API**

All FMOD functions dealing with file names and paths now accept UTF-8 encoded strings. All arguments that take or return UTF-16 encoded strings have been removed, as have flags that indicate an UTF-16 string has been passed.

UTF-8 encoded strings can be used to access file paths containing non-ASCII characters on Windows, PS3, PS4, PS Vita, XBox One, iOS, Android, Mac, Linux, Windows Phone, and Windows Store.
Added AAC Support for Android

It is now possible to load AAC files on Android 4.2 or newer devices. This can make use of any hardware accelerated decoding provided by the platform.
Improved Logging Output

For greater control of the logging version of FMOD, i.e. fmodL / fmodstudioL you can now use FMOD::Debug_Initialize to specify both logging level and destination. By default all output will go to the TTY / stderr / logcat / platform specific debug location. Using this new API you can also send output to file or callback for integration into custom logging systems.
DSP Reset Callback Behavior Clarified

The **FMOD_DSP_RESET_CALLBACK** documentation has been updated to make it clear that it should leave public parameters unchanged.
Firelight Technologies FMOD Studio API
What's New in 1.04

This section describes the major features introduced in the 1.04 release. See the Detailed Revision History for information regarding each patch release.
Studio Profiling

FMOD Studio now supports profiling. Profiling sends event, parameter, and bus information across the network to FMOD Studio, allowing the profiling session to be visualised and replayed. Profiling is supported automatically when the `FMOD_STUDIO_INIT_LIVEUPDATE` flag is used.

Note that the existing low level profiler is still supported, and can be used for in-depth analysis of the DSP graph when using the Studio API or the Low Level API. The low level profiler is provided in the Windows or Mac Programmer's API package as the "FMOD Profiler" executable. The low level profiler is supported automatically when either the `FMOD_STUDIO_INIT_LIVEUPDATE` or the low level `FMOD_INIT_PROFILE_ENABLE` flag is used.
Audio Tables

FMOD Studio now supports adding audio tables to banks. These are designed to hold large numbers of sounds for use cases like localized VO. Each entry in the table is identified by a key string, and the mapping from key to sound can be changed by loading a different audio table.

To use an audio table entry, look it up with `Studio::System::getSoundInfo` and then pass the resulting information to `System::createSound`. This sound will contain a subsound that can be used with `FMOD_STUDIO_EVENT_CALLBACK_CREATE_PROGRAMMER_SOUND` or played back directly. Note that the bank containing the audio table must be loaded for the lookup to succeed.
Sub-Events Deprecated

Sub-events are now deprecated. This means Studio::EventInstance::createSubEvent will be removed in a future release. Note that Event Sounds are still supported, and are the preferred mechanism for event nesting.
Event Callback Changed

`FMOD_STUDIO_EVENT_CALLBACK` now has an extra `FMOD_STUDIO_EVENTINSTANCE*` parameter. This simplifies callback implementation, as you no longer need to check the callback type in order to get the event.
Deprecated Functions Removed

Some functions that were deprecated have now been removed:

- Studio::EventInstance::getLoadingState - use Studio::EventDescription::getSampleLoadingState instead
- Studio.Factory.System_Create (C# only) - use Studio.System.create instead
- Studio.System.init (C# only) - use Studio.System.initialize instead
**PS3 Support**

FMOD Studio now officially supports PS3.
WiiU Support

FMOD Studio now officially supports WiiU.
Linux Support

FMOD Studio now officially supports Linux.
Windows Phone 8.1 Support

FMOD Studio now officially supports Windows Phone 8.1.
Firelight Technologies FMOD Studio API
What's New in 1.03

This section describes the major features introduced in the 1.03 release. See the Detailed Revision History for information regarding each patch release.
**Threading Changes**

FMOD Studio has support for multi-threading in both low-level and Studio API. The Studio API supports issuing commands from any number of threads, and by default now does all execution in its own thread. The Low Level API now supports thread safety by default. FMOD Studio and Low Level threading and thread safety can be disabled with new initialization flags. See the [Threads and Thread Safety](#) section for more information.
**Handle and Studio API Changes**

As part of the threading changes, the Studio API handle system has been changed. The Studio C++ API now matches the Low level C++ API which has a pointer-based system. This means existing code will have to change from directly using Studio types to retrieving and calling API with a pointer style. The underlying handle system has been rewritten to be thread-safe and more efficient. This change also means that the C API now properly deals with invalid handles safely, and that the C API and C++ API types are equivalent and can be cast between each other. See the [Handle System](#) for more information.
Recording and Playback

All Studio API calls can be recorded to a file and played back at a later time. This is intended for debug functionality. See Studio::System::startRecordCommands, Studio::System::stopRecordCommands and Studio::System::playbackCommands for more details.
Studio Enumeration Functions

FMOD Studio enumeration functions have been changed to a getCount/getList style which is thread-safe and accessible by the C interface. More functions have been added to enumerate banks, events, and mixer strips. See

Studio::System::getBankCount, Studio::System::getBankList,
Studio::Bank::getEventCount, Studio::Bank::getEventList,
Studio::Bank::getMixerStripCount, Studio::Bank::getMixerStripList,
Studio::EventDescription::getInstanceCount, and
Studio::EventDescription::getInstanceList.
Events in multiple banks

FMOD Studio now supports duplicate events in multiple banks. Studio resources are reference counted and loading will skip the event data that has already been loaded in an existing bank.
Error Callback

All Low Level API and Studio API errors can now be intercepted by registering a user callback. The callback includes the function that generated the error, the instance data, and the optional function parameter arguments. This is particularly useful for Studio API commands which have been deferred to run on the Studio asynchronous thread. See System::setCallback for more information.
PS Vita support

FMOD Studio now officially supports PS Vita.
Windows Store Applications support

FMOD Studio now officially supports Windows Store Applications for both x86 and ARM devices.
Additional Studio features

FMOD Studio now supports sounds placed on transition timelines, transition marker probability, and loop region probability.
Documentation

The Studio API documentation has now improved and extended.
Firelight Technologies FMOD Studio API
What's New in 1.02

This section describes the major features introduced in the 1.02 release. See the Detailed Revision History for information regarding each patch release.
Android Platform

FMOD Studio now officially supports the Android platform.
Unity Support

FMOD Studio has added support for the Unity game engine via a native plugin.
New Bank Loading API

The existing loadBank API has been renamed to Studio::System::loadBankFile, and two extra versions have been added. Studio::System::loadBankMemory allows loading from a memory location. Studio::System::loadBankCustom supports loading banks with bank-specific callbacks to do the reading.
Memory and performance optimisations

This revision includes a number of memory and CPU optimisations to both low-level and the Studio API.
**Documentation**

FMOD Studio now ships with documentation that covers the full API.
Firelight Technologies FMOD Studio API
What's New in 1.01

This section describes the major features introduced in the 1.01 release. See the Detailed Revision History for information regarding each patch release.
Sends

Support has been added for Send DSP connections to send sound data to different areas of the mix graph.
**Fade Points**

Support has been added for setting sample-accurate fade ramp points. The fade points allow Studio to schedule sound volume gradients to occur in future mixes.
Studio API

A number of additional Studio API functions have been added. See the revision.txt for more details.
Firelight Technologies FMOD Studio API
Windows Specific Starter Guide

SDK Version

FMOD is compiled using the following tools.

- **Visual Studio** - version 2012 targeting platform toolset v110_xp.
Compatibility

FMOD supports the below architectures back to Windows XP.

- **x86** - optimized with SSE3.
- **x86_64** - optimized with SSE3 (and AVX if detected at runtime).
Libraries

The provided libs are import libraries which require the corresponding DLL to be present at runtime.

FMOD Low level library

- /api/lowlevel/lib/fmod64_vc.lib - Release 64-bit binary for production code (requires fmod64.dll at runtime).
- /api/lowlevel/lib/fmodL64_vc.lib - Release 64-bit binary with logging enabled for development (requires fmodL64.dll at runtime).

FMOD Studio Runtime library (used in conjunction with low level library)

- /api/studio/lib/fmodstudio64_vc.lib - Release 64-bit binary for production code (requires fmodstudio64.dll at runtime).
- /api/studio/lib/fmodstudioL64_vc.lib - Release 64-bit binary with logging enabled for development (requires fmodstudioL64.dll at runtime).
Before calling any FMOD functions it is important to ensure COM is initialized. You can achieve this by calling `CoInitializeEx(nullptr, COINIT_APARTMENTTHREADED)` on each thread that will interact with the FMOD API. This is balanced with a call to `CoUninitialize()` when you are completely finished with all calls to FMOD.

If you fail to initialize COM, FMOD will perform this on-demand for you issuing a warning. FMOD will not uninitialize COM in this case so it will be considered a memory leak.

To ensure correct behavior FMOD assumes when using the WASAPI output mode (default for Windows Vita and newer) that you call output related functions from your UI thread. This ensures that any platform specific dialogs that need to be presented can do so. This recommendation comes from the `IAudioClient` interface docs on MSDN which state:

In Windows 8, the first use of `IAudioClient` to access the audio device should be on the STA thread. Calls from an MTA thread may result in undefined behavior.
Known Issues

No known issues, please contact support@fmod.com if you discover a problem.
Firelight Technologies FMOD Studio API
Performance Reference

This document is a companion for the performance tutorial document and serves as a quick reference of facts targeting this platform.
Format Choice

Each compression format provided in FMOD has a reason for being included, the below list will detail our recommendations for this platform. Formats listed as primary are considering the best choice, secondary formats should only be considered if the primary doesn't satisfy your requirements.

- **Vorbis**: Primary format for all sounds.
- **FADPCM**: Secondary format if Vorbis CPU usage is too high for low spec machines.
- **PCM**: Not recommended.
- **XMA**: Unavailable.
- **AT9**: Unavailable.
**Voice Count**

To give developers an idea about the costs of a particular format we provide synthetic benchmark results. These results are based on simple usage of the FMOD Studio API using recommended configuration settings.

**Settings**

- **Voice count:** 64
- **Sample rate:** 48KHz
- **Speaker mode:** Stereo
- **DSP block size:** 1024 samples

**Test Device**

- **CPU:** Intel Core 2 Duo E6600 @ 2.4 GHz
- **OS:** Windows XP

**Results**

- **DSP with Vorbis:** 22.2% (+/- 1.6%)
- **DSP with FADPCM:** 5.5% (+/- 0.1%)
- **DSP with PCM:** 3.3% (+/- 0.1%)
- **Update at 60 FPS:** 0.8% (+/- 0.2%)
Firelight Technologies FMOD Studio API
Detailed Revision History

06/12/17 1.10.02 - Studio API minor release (build 92217)

Features:

- **LowLevel API - UWP** - Added Windows Sonic support.
- **LowLevel API - UWP** - Added support for setting thread affinity via `FMOD_UWP_SetThreadAffinity`.

Fixes:

- **Studio API** - Fixed Quietest stealing behavior so new events don't start if they are quieter than all currently playing events.
- **Studio API** - Fixed buses that have reached their Max Instances limit incorrectly preventing their input buses from applying their stealing behavior.
- **Studio API** - Fixed a crash caused by wav files with invalid loop points.
- **LowLevel API** - Fixed recording devices still being marked as connected when switching to `FMOD_OUTPUTTYPE_NOSOUND`.
- **LowLevel API - UWP** - Made device reset / unplug behavior more robust against failure.
- **LowLevel API - Xbox One** - Fixed WASAPI init error if WinSonic was attempted first.
- **FSBank API** - Fixed crash in 64bit version when encoding low sample rate mono sounds as Vorbis with low quality.
- **Unity** - PlayOneshot and PlayOneshotAttached now log a warning when the event is not found, rather than throwing an exception.

Notes:

- Added Resonance Audio plugin version 1.0, this plugin represents the continuation of the Google VR plugin under new branding. The Google VR plugin is now considered deprecated in favour of Resonance Audio,
consider migrating to the new plugin as the GVR plugin will be removed in FMOD 1.11.

- *Xbox One* - Now built with June 2017 QFE 8 XDK.
Features:

- **UE4** - Expose Bus::stopAllEvents to be useable through blueprints.
- **Unity** - Event length displayed in event browser for one shot events.

Fixes:

- **Studio API** - Fixed modulation on plugin instruments using the parent event's lifetime rather than the instrument's lifetime.
- **Studio API** - Fixed a live update issue where asset files would not reload after being moved on disk.
- **Studio API** - Fixed a bug which caused sounds to pan incorrectly when using **FMOD_INIT_3D_Righthanded**.
- **Studio API** - PS4 - Fixed excessive binary size caused by inclusion of debug symbols.
- **LowLevel API** - Fixed potential crash when re-initializing FMOD::System after failure.
- **LowLevel API** - Fixed compatibility issues with Windows XP.
- **LowLevel API** - Android - Fixed exported symbols to avoid issues with unwinder.
- **LowLevel API** - Android - Automatic output mode selection will now choose AudioTrack rather than OpenSL if Bluetooth is enabled to avoid stuttering.
- **LowLevel API** - Xbox One - Ensure WinSonic internal threads are assigned to the same core as the FMOD mixer.
- **FSBank API** - Fixed crash when encoding very long audio files.
- **UE4** - Integration now handles application interruption by pausing and resuming the mixer.
- **UE4** - Fixed SetEvent not setting or using new event.
- **UE4** - Fixed Xbox One thread affinity struct setup.
- **UE4** - Removed engine version ifdef's.
- **UE4** - Fixed integration attempting to set the initial value of built-in parameters.
- **Unity** - Fixed compatibility for Unity 5.0 & 5.1.
- **Unity** - Added check to see if any banks have been loaded before trying to pause.
- **Unity** - Allow StringHelper to fast return if string is null.

**Notes:**

- Updated Google VR plugin to version 0.6.1.
- **Studio API** - [Studio::EventInstance::set3DAttributes](#) and [Studio::System::setListenerAttributes](#) will now return [FMOD_ERR_INVALID_PARAM](#) if the forward or up vectors are zero. In logging builds warnings will be logged if the forward and up vectors are not orthonormal.
- **LowLevel API** - The convolution reverb effect will not accept impulse response data if the system is not using a power-of-two DSP buffer size. Windows Sonic currently requires a DSP buffer size of 480 making it incompatible with convolution reverb until this requirement is lifted.
- **PS4** - Now built with SDK 5.008.001.
- **Unity** - Exposed editor script classes for in-game FMOD objects as public.
- **Unity** - Exposed StudioEventEmitter's EventInstance as public to allow easier integration with custom plugins.
19/09/17 1.10.00 - Studio API major release (build 90329)

Features:

- LowLevel API - Added `FMOD_MAX_SYSTEMS` constant, currently 8.
- LowLevel API - Exposed `FMOD_DSP_FADER_GAIN` on `FMOD_DSP_TYPE_FADER`.
- LowLevel API - Added `FMOD_SPEAKERMODE_7POINT1POINT4` speaker mode which includes four height speakers to be used with Windows Sonic output.
- LowLevel API - Added `FMOD_DSP_PAN_2D_HEIGHT_BLEND` parameter to `FMOD_DSP_TYPE_PAN` that allows mixing ground speaker signal into the height speakers and vice versa.
- LowLevel API - Windows & Xbox One - Added Windows Sonic output plugin to support rendering multichannel (with height) speaker mode 7.1.4 as well as dynamic objects via `FMOD_DSP_TYPE_OBJECTPAN`.
- LowLevel API - PS Vita - Switched `FMOD_PSVita_SetThreadAffinity` to accept a mask instead of a core number to allow floating threads.
- LowLevel API - Added `FMOD_OUTPUT_REQUESTRESET` to `FMOD_OUTPUT_STATE` to allow output plugins to request they be reset by the System.

Fixes:

- Studio API - Sequential multi and scatterer instruments now track their current playlist entry on a per-event-instance basis, rather than globally.
- UE4 - Removed all reference to Oculus, now that Oculus functions as all other FMOD Studio plugins.
- UE4 - Removed all legacy UE4 version code, if you are not using UE4.16 the plugin will not work without making changes to source code.
- UE4 - Overhauled details interaction in editor and improved usability. Grouped all FMOD functionality together, added Parameters, and removed unnecessary information from attenuation/occlusion.
- Unity - Removed garbage allocations from C# wrapper.
Notes:

- Updated Google VR plugin to version 0.6.0.
- **LowLevel API** - `FMOD_DSP_TYPE_ENVELOPEFOLLOWER` is now deprecated and will be removed in a future release.
- **LowLevel API** - Increment Plugin API version for Output plugins. Dynamic library Output Plugins must be rebuilt.
- **Studio API** - Changed the behaviour of nested events to stop when idle even when there are instruments on parameters. This makes nested events match the behaviour of top level events. Events which depend on the old behaviour need to be manually fixed up by (for example) adding a sustain point to the nested events timeline.
- **Android** - Logging version will now produce proper crash stacks but due to binary size increase the release version will continue to not.
- **Xbox One** - Removed "acp" and "feeder" from FMOD_DURANGO_THREADAFFINITY. Both threads were removed in previous versions and setting them did nothing.
- **UE4** - AudioComponents using occlusion from previous versions are NOT compatible with this version. Occlusion and Attenuation now do not rely on UE4 structs.
11/09/17 1.09.08 - Studio API minor release (build 90162)

Fixes:

- LowLevel API - Fixed extraneous logging.
Features:

- **UE4** - Cache dsp used for occlusion lowpass effect & add support for use of Multiband EQ (lowpass on band A only).
- **UE4** - FMODAudioComponent now reuses event instances until the object is no longer needed or Release() is called.
- **UE4** - Added support for UWP.
- **Unity** - Added support for Unity v2017.1.
- **Unity** - Added a button in the FMOD menu for Refreshing Bank files.

Fixes:

- **Studio API** - Fixed scatterer sounds being processed by FMOD::Geometry.
- **Studio API** - Fixed multi-stream events playing out of sync (introduced in 1.09.01).
- **LowLevel API** - Fixed ChannelControl::setDelay being ignored if addDSP was called immediately after it.
- **LowLevel API** - Fixed potential crash if calling Channel::setPosition soon after System::playSound with paused as true.
- **LowLevel API** - Fixed click on Channel::setPaused(false) caused by a non-zero Channel::setPosition after System::playSound with paused as true.
- **LowLevel API** - Fixed potential crash if calling System::getRecordPosition while disconnecting an audio device.
- **LowLevel API** - Fix FMOD_ACCURATETIME not looping mod/s3m/xm/it files properly, and midi files not looping properly without the flag.
- **LowLevel API** - Fixed crash when attempting to load invalid VST files.
- **UE4** - Fix compile error by adding categories to AnimNotify vars.
- **Unity** - Switch - Fixed "unknown pointer encoding" error when an exception occurs.
- **Unity** - Fix plugin path for UWP builds.
- **Unity** - Fixed possible crash when using GoogleVR plugin.
- **Unity** - Fix EventEmitter SetParameter not working unless parameter had an initial value set in editor.
Notes:

- **Studio API** - Reduced memory usage for events with a small number of instances.
- **LowLevel API** - FMOD_CREATESOUNDEXINFO::initialseekposition will now wrap if the value given is longer than the FMOD::Sound length.
- **LowLevel API** - Added documentation to the top of fmod_codec_raw to be more instructional for plugin writers.
- **UE4** - Added support for UE4.17.
- **Unity** - Device specific errors will now cause only a single exception to be thrown and the integration will assume no-sound mode.
- **Switch** - Now built with SDK 1.7.0.
- **Xbox One** - Now built with March 2017 QFE 3 XDK.
06/07/17 1.09.06 - Studio API minor release (build 88495)

Features:

- Studio API - Improved performance when a large number of EventInstances have been created but not started.
- LowLevel API - HTML5 - Performance increased by 10%.

Fixes:

- Studio API - Fixed a bug where an asynchronous looping multi instrument would stop selecting new playlist entries after playing a nested event which itself contains an asynchronous looping multi instrument.
- Studio API - Fixed a bug where a parameter could trigger parameter instruments before having its value updated by modulators when restarting an event instance.
- Studio API - Fixed incorrect automation interpolation on transition timelines with lead-out regions.
- LowLevel API - Fixed DSPs with sidechain inputs incorrectly going idle when the main input is idle but the sidechain input is not.
- LowLevel API - Fixed the compressor DSP not playing out its release correctly when its inputs are idle.
- LowLevel API - Remove main thread stall from `System::playDSP` (or playing a generator DSP via Studio API).
- UE4 - Sequencer integration now supports previewing event playback in an editor viewport by using the Sequencer transport controls. A Sequencer section has been added to the documentation.
- UE4 - Added AreBanksLoaded function to FMODStudioModule.
- Unity - Events in EventBrowser window now in alphabetical order.
- Unity - Setting parameters on StudioEventEmitter no longer generates garbage.

Notes:

- LowLevel API - Added checking to `System::mixerResume` to ensure called
from the same thread as System::mixerSuspend.

- *Xbox One* - Now built with October 2016 QFE 3 XDK.
08/06/17 1.09.05 - Studio API minor release (build 87666)

Features:

- **LowLevel API - Switch** - Added support for HTC sockets to allow communications between FMOD tools and runtime via target manager. Enable using FMOD_NX_SetHTCSEnabled(TRUE).
- **LowLevel API - Xbox One** - Added support for System::attachChannelGroupToPort with FMOD_DURANGO_PORT_TYPE_MUSIC.

Fixes:

- **Studio API** - Fixed FMOD_ERR_INTERNAL returned when loading old bank files containing transition timeline automation of instrument volumes.
- **Studio API** - Fixed bug where very short instruments would not play when cross fading.
- **Studio API** - Made changes to the logic in EventDescription::isOneShot so that it consistently returns true for events which are guaranteed to finish without intervention, and false for events which may play indefinitely.
- **LowLevel API** - Fixed ChannelControl::setMixLevelsInput not working and updated docs.
- **LowLevel API** - Fixed Channel::setLoopCount not working with very small streams.
- **LowLevel API** - Stricter error checking when loading IMA ADPCM wav files to prevent a potential crash from malformed data.
- **LowLevel API** - Fixed potential crash in ChannelGroup operations when a Channel failed to play on it with FMOD_ERR_SUBSOUNDS.
- **LowLevel API** - Fixed Convolution reverb panning a mono IR with a stereo input incorrectly.
- **LowLevel API** - Fixed race conditions when setting FMOD_DSP_SEND_RETURNID.
- **LowLevel API** - Fixed a crash with some MOD/S3M/XM/IT files. Introduced in 1.09.00.
- **LowLevel API** - System::setDSPBufferSize will round the requested buffer
size up to the closest multiple of 4 to prevent a crash when sending metering data to studio.

- **LowLevel API - PS4** - Fixed GPU compute compatibility issue with SDK 4.508.001. GPU compute is now re-enabled.
- **LowLevel API - Switch** - Reduced thread priority to avoid conflict with Profiler.
- **LowLevel API - Windows** - Fixed ASIO output mode failing to initialize if the devices requires a buffer size of 2048 samples.
- **Unity** - Fixed bank directory path separators when developing across OSX & Win.
- **Unity** - Fixed simulated Android devices producing no sound.
- **Unity** - BankLoadException now display error message correctly.
- **Unity** - Fixed bank loading and unloading refcount accuracy.
- **Unity** - Fixed Mac editor attempting to load Linux plugins when building for Linux platform.
- **Unity** - Improved detection of 3D Event Instances that haven't had their position set yet.
- **UE4** - Fixed integration working with UE4's IWYU non-monolithic header system, for now the integration is still using the old PCH system.
- **UE4** - Added new native AnimNotify class, old one didn't work on code projects.
- **UE4** - Sequencer integration. FMOD events can be started and stopped and event parameters can be controlled by adding custom tracks to sequencer.
- **UE4** - Fixed max vorbis codecs not being set correctly.
- **UE4** - Fixed file readers being accessed from multiple threads.

**Notes:**

- **UE4** - Added support for UE4.16.

**Notes:**

- Updated Google VR plugin to version 0.4.0, please note there is a known crash when loading the plugin on Windows XP, Google are aware and investigating.
10/04/17 1.09.04 - Studio API minor release (build 86084)

Fixes:

- **Studio API** - Fixed delayed playback on streaming sounds in events (introduced in 1.09.03).
- **Studio API** - Fixed AHDSR release not working on single sounds shorter than 100 milliseconds.
- **Studio API** - Fixed EventDescription::is3D returning true for events that only have 2D panners.
- **Studio API** - Set programmer sounds to `FMOD_LOOP_NORMAL` internally if they were not created that way.
- **Studio API** - Fixed regression introduced in 1.09.00 which allowed events to play at the origin before 3d attributes were updated.
- **Studio API** - Fixed issue where reverb tail would cut off when all events on a bus finished playing.
- **LowLevel API** - Fixed `FMOD_DSP_TRANSCIEIVER` making channels audible that weren't supposed to be (introduced with glitch fix in 1.09.03).
- **LowLevel API** - Fixed loop clicks on PCM sounds if using `FMOD_DSP_RESAMPLE_CUBIC` or `FMOD_DSP_RESAMPLE_SPLINE`.
- **LowLevel API** - Fixed FMOD_ADVANCEDSETTINGS::resamplerMethod being ignored.
- **LowLevel API** - Fixed plugin unloading for multi-description libraries potentially failing depending on how it's unloaded.
- **LowLevel API** - Fixed stream glitch when going virtual then resuming.
- **LowLevel API** - Fixed virtual voices losing their loop/2d/3d status, and not staying virtual if `Channel::setMode` was used. Introduced in 1.09.00.
- **LowLevel API** - Fixed `FMOD_UNIQUE` not being accepted if `Channel::setMode` or `Sound::setMode` was used. (it could be successfully used via createSound/createStream).
- **LowLevel API** - Fixed rare crash in mixer, introduced 1.09.00.
- **LowLevel API** - Switch - Fixed FMOD_NXTHREADAFFINITY so cores can be ORd together to form a mask.
- **LowLevel API** - **PS4** - GPU compute disabled due to an incompatibility with SDK 4.508.001.
- **LowLevel API - Windows/Mac** - Re-enable SRS downmixer 80Hz high pass filter by default. Add `FMOD_INIT_DISABLE_SRS_HIGHPASSFILTER` init flag to disable it.
- **FSBank API** - Fixed PS Vita AT9 encoder not working with currently available Sony library.
- **FSBank API** - Fixed full scale 32bit float wav files encoding incorrectly.

**Notes:**

- **Studio API** - FMOD expects programmer sounds to be created with `FMOD_LOOP_NORMAL`. This is now specified in the `FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES` documentation.
- **FSBank API** - Building an FSB with PS Vita AT9 encoding now requires 64bit.
- **PS4** - Now built with SDK 4.508.001.
- **Switch** - Now built with SDK 0.12.17.
- **Xbox One** - Now built with October 2016 QFE 2 XDK.
20/03/17 1.09.03 - Studio API minor release (build 85359)

Features:

- *LowLevel API* - Updated the /examples/dsp_custom example to include a lot more functionality including parameters, and capture of wave data.
- *LowLevel API* - Add fmod_reduced.js for reduced functionality, but also reduced size.
- *LowLevel API - Switch* - Added support for setting thread affinity.
- *Studio API* - Reduced size of fmodstudio.js and .mem files.

Fixes:

- *Studio API* - Fixed event doppler settings not being applied to sounds spawned by scatterer.
- *Studio API* - Fixed bus and VCA handles not being set up properly in Studio::Bank::getBusList and Studio::Bank::getVCAList.
- *Studio API* - Fixed crashes caused by stopping events that are routed into a bus with instance limiting enabled when they are in the FMOD_STUDIO_PLAYBACK_STARTING state.
- *Studio API* - Fixed tempo and marker event callbacks not being fired when the timeline cursor is in the lead-in or lead-out region of a transition timeline.
- *LowLevel API* - Fixed glitches with Transceiver DSP after inputs go idle.
- *LowLevel API* - Fix Channel::setPosition(pos, FMOD_TIMEUNIT_MODORDER) not working when playing paused.
- *LowLevel API* - Fixed short streams created as non-looping then switched to looping via the Channel API not looping seamlessly.
- *LowLevel API* - Fixed DSP plugin version >= 109 data parameters other than 3D attributes not applying if FMOD_INIT_3D_RIGHTHANDED is used.
- *LowLevel API* - Fixed rare crash in FMOD Panner DSP. Introduced in 1.09.00.
- *LowLevel API* - Re-Fix MOD/S3M/XM/IT file crash with samples that have 0 length, for 1.09 only.
- **LowLevel API** - Fixed potential memory leak if `System::init` returned an error.
- **LowLevel API** - Fix `FMOD_ACCURATETIME` not looping a mod file properly, and not seeking correctly with `FMOD_TIMEUNIT_MODORDER`.
- **LowLevel API** - **HTML5** - Loading FSB sounds did not work properly.
- **LowLevel API** - **Windows** - Fixed 5.1->stereo SRS downmix causing lack of bass.
- **LowLevel API** - **Windows** - Fix `FMOD_INIT_PREFER_DOLBY_DOWNMIX` not working.
- **FSBank API** - Fix crash if `FSBANK_INIT_GENERATEPROGRESSITEMS` is not used.
- **Unity** - Removed error when plugin field is added but empty.
- **UE4** - Removed error when plugin field is added but empty.

**Notes:**

- **UE4** - Added support for UE4.15.
15/02/17 1.09.02 - Studio API minor release (build 84334)

Fixes:

- *Unity* - Remove ifdef from EnforceLibraryOrder as it isn't harmful for static lib platforms to call GetStats.
- *UE4* - Occlusion can now use Multiband EQ instead of Lowpass filter.
- *LowLevel API* - Fix crash when connecting to FMOD Profiler.exe and there is a circular connection
- *LowLevel API* - Fix glitches with Transceiver DSP after inputs go idle.
- *LowLevel API* - Fix `Channel::setPosition(pos, FMOD_TIMEUNIT_MODORDER)` not working when playing paused.
Features:

- **LowLevel API** - Added FMOD_DSP_STATE_FUNCTIONS::getlistenerattributes to the DSP plugin API to query the current listener attributes.
- **Unity** - Added support for Rigidbody2D in 3D attribute settings and integration scripts.
- **Unity** - Added support for Object Enable/Disable on EventEmitter and ParameterTrigger scripts.
- **UE4** - Added GetLength function for blueprints that returns the event length in milliseconds.
- **UE4** - Improved in editor profiling stats.

Fixes:

- **Unity** - Fixed compatibility with Unity 4.6 & 4.7 for OSX and IOS.
- **Unity** - Fixed "file not found" error when settings asset is corrupt
- **Unity** - Fixed set3DAttributes ambiguity.
- **Unity** - Fixed not being able to copy ReadOnly banks into StreamingAssets.
- **Unity** - Fixed event instance leak in Unity PlayOneshotAttached not releasing.
- **Unity** - Specified version to check for WiiU BuildTarget for early Unity5.
- **UE4** - Fixed XboxOne delayload error now that UE4 handles it.
- **UE4 - Android** - Added ARM64 support.
- **Studio API** - AHDSR modulator curve shapes now work correctly. In previous versions the envelope was interpolated linearly regardless of the shape displayed in the UI.
- **Studio API** - Fixed looping single sounds in an async multi sound being cut-off when the multi sound is un-triggered.
- **LowLevel API** - Fixed Channel::setReverbProperties not resetting reverb connection to a new tail DSP when turning wet mix off and on.
- **LowLevel API** - If the system is initialized in right-handed mode, FMOD will now swap to left-handed when passing attributes into plugins. This
only applies to plugins rebuilt against this version, old plugins remain unswapped.

- **LowLevel API** - Fix MOD/S3M/XM/IT file crash with samples that have 0 length.
- **LowLevel API** - Fixed some potential crashes when running out of memory, these will correctly return `FMOD_ERR_MEMORY` now.
- **LowLevel API** - **Win** - Fixed WASAPI recording device enumeration taking a couple of seconds after `System::init` before being correct.
- **LowLevel API** - **Win** - Fixed potential error returned from `System::update` if device is unplugged.
- **LowLevel API** - **MIDI** - Fixed Sound::set/getMusicChannelVolume referring to wrong track indices rather than just a normal 0-15 track index.
- **LowLevel API** - **MIDI** - Fixed `Channel::setPosition` causing loud drum bang noise after seek

**Notes:**

- **Studio API** - When loading legacy banks with looping sounds nested in multi sounds, the multi sound is set to cut-off all sounds when untriggered, including non-looping sounds. This is a change in behaviour compared to earlier versions where only looping sounds were cut-off.
- **LowLevel API** - DSP plugin API version has been increased, for maximum compatibility plugin writers should only rebuild against this version if they need the getlistenerattributes feature. Old plugins are still supported.
- **Switch** - Now built with SDK 0.12.10
01/12/16 1.09.00 - Studio API major release (build 82164)

Important:

- Added support for the Nintendo Switch platform.

Features:

- **Studio API** - Added support for multiple listener weights, with `Studio::System::setListenerWeight` and `Studio::System::getListenerWeight`. This allows listeners to be faded in or out smoothly.
- **Studio API** - Profiling uses less CPU and memory.
- **Studio API** - Added `FMOD_STUDIO_INIT_LOAD_FROM_UPDATE` to drive all loading from `Studio::System::update` rather than the bank and resource loading threads. Mainly used in non-realtime situations.
- **Studio API** - Added ability to run replays at faster than real time speed using `FMOD_STUDIO_COMMANDREPLAY_FAST_FORWARD`.
- **Studio API** - Added `Studio::EventInstance::setReverbLevel` and `Studio::EventInstance::getReverbLevel`.
- **Studio API** - Support for automation of modulator properties.
- **LowLevel API** - Improved memory use when loading sounds with names.
- **LowLevel API** - Added new efficient multiband equalizer (`FMOD_DSP_TYPE_MULTIBAND_EQ`) featuring 5 toggleable bands with variable rolloff low/high pass, shelf, peaking, band-pass, all-pass and notch modes.
- **LowLevel API** - Added logging function to the DSP plugin API. Use `FMOD_DSP_LOG` helper macro from a DSP callback.
- **LowLevel API** - Added `FMOD_DSP_STATE_FUNCTIONS::getuserdata` to the DSP plugin API. Use `FMOD_DSP_GETUSERDATA` helper macro from a DSP callback.
- **FSBank API** - Added ability to pass NULL as the FSB file name for `FSBank_Build` causing the FSB to be built in memory. Use `FSBank_FetchFSBMemoory` to get access to the data once the build has completed.
Fixes:

- *Studio API* - Fixed incorrect playback volume for instruments and playlist items whose volume property is set to a non-zero dB value.

Notes:

- *LowLevel API* - Incremented Plugin SDK version for DSP plugins. Dynamic library plugins built with earlier versions will continue to load.
- *LowLevel API* - Incremented `FMOD_CODEC_WAVEFORMAT` version. Codecs that provide names must keep the name memory persistent for the lifetime of the codec.
- *LowLevel API* - Increment Plugin API version for Output plugins. Dynamic library Output Plugins must be rebuilt.
- *LowLevel API* - `FMOD_DSP_TYPE_LOWPASS`, `FMOD_DSP_TYPE_LOWPASS_SIMPLE`, `FMOD_DSP_TYPE_HIGHPASS`, `FMOD_DSP_TYPE_HIGHPASS_SIMPLE` and `FMOD_DSP_TYPE_PARAMEQ` are considered deprecated and will be removed in the future. Use the new `FMOD_DSP_TYPE_MULTIBAND_EQ` instead which has the performance of "simple" effects with full featured quality.
- *LowLevel API* - Changed reverb wet mix to send from the fader DSP of a ChannelGroup. Previously it sent from the head DSP. Now effects placed pre-fader will apply to the signal sent to the reverb, while effects placed post-fader will not.
- *LowLevel API* - ChannelGroup reverb wet level is now scaled by the group's effective audibility.
- *LowLevel API* - ChannelGroup reverb no longer automatically disables reverb on its child channels when the wet level is set to non-zero.
- *LowLevel API* - `Channel::setReverbProperties` now allows setting the wet level before the specified reverb instance has been created.
- *LowLevel API* - `ChannelControl::addDSP` and `ChannelControl::removeDSP` manage standard DSP connections (`FMOD_DSPCONNECTION_TYPE_STANDARD`) to maintain the mixer hierarchy. Other connection types (`FMOD_DSPCONNECTION_TYPE_SIDECHAIN`, `FMOD_DSPCONNECTION_TYPE_SEND_SIDECHAIN`, and now
**FMOD_DSPCONNECTION_TYPE_SEND** are left undisturbed.

- **LowLevel API** - **FMOD_INIT_MIX_FROM_UPDATE** will now directly execute the mixer from **System::update** instead of triggering the mix to happen in another thread.
- **Studio API** - Incremented bank version, requires runtime 1.09.00 or later.
- **Studio API** - Latest runtime supports loading old bank versions from 1.03.00.
- **Studio API** - Studio::Bus::setFaderLevel, Studio::Bus::getFaderLevel, Studio::VCA::setFaderLevel and Studio::VCA::getFaderLevel is now called **Studio::Bus::setVolume, Studio::Bus::getVolume, Studio::VCA::setVolume** and **Studio::VCA::getVolume**.
- **Studio API** - Studio::EventInstance::getVolume, Studio::EventInstance::getPitch, Studio::EventInstance::getParameterValue, Studio::EventInstance::getParameterValueByIndex, Studio::Bus::getVolume, Studio::VCA::getVolume now have an additional argument to get the final value which includes automation and modulation.
- **Studio API** - The required alignment for **Studio::System::loadBankMemory** has been added as the constant **FMOD_STUDIO_LOAD_MEMORY_ALIGNMENT**.
- **Studio API** - Event instances now disable reverb on their internal channels, for all global reverb instances. Previously only did so for reverb instance 0. Use **Studio::EventInstance::setReverbLevel** to control the reverb mix for the whole event instance.
- **Studio API** - Disconnected sidechain modulators are now inactive. Previous behavior was to fall back to monitoring the event's master track output.
- **LowLevel API** - **FMOD_DSP_RELEASE_CALLBACK** is now called from the main thread
- **LowLevel API** - Unimplemented ChannelControl::overridePanDSP function has been removed, along with **FMOD_CHANNELCONTROL_DSP_PANNER** enum value.
- **LowLevel API** - **PS3** - Removed old opt-in FIOS support via **FMOD_PS3_EXTRADRIVERDATA**. New recommended approach is to use **FMOD_FILEASYNCREAD_CALLBACK** and set appropriate deadlines based on **FMOD_ASYNCREADINFO::priority**.
- **LowLevel API** - **Xbox One** - Optimized XMA decoding performance and removed the ACP thread.
- **LowLevel API** - **PS4** - Improved AT9 decoding performance.
- **FSBank API** - Added support for source data as a memory pointer instead of file name.
• *Documentation* - Some FMOD_DSP_PAN_SURROUND enums changed to FMOD_DSP_PAN_2D for clarity.
01/12/16 1.08.15 - Studio API minor release (build 82163)

Features:

- **PS4** - Add support for `System::getOutputHandle`, to return sce port handle.

Fixes:

- **UE4** - Fix missing plugin error when building on Mac.
- **Unity** - Fixed OSX working with unified library.
- **Unity** - Fixed WiiU copying banks error.
- **Unity** - Fixed Xbox one dll meta files missing platform target.
- **Unity** - Fixed duplicate dll copying build error on some platforms.
- **Unity** - Added null check to stop error being thrown when no event assigned.
- **Unity** - Fix in editor out of bounds exception in RuntimeManager.
- **LowLevel API** - Allow `DSP::setParameterData` with null data and 0 length to free convolution reverb impulse response data.
- **LowLevel API** - Fixed short looping streams playing some of the start when switched to non-looping via the Channel API.
- **LowLevel API** - Fixed `FMOD_CREATESOUNDEXINFO::pcmsetposcallback` getting wrong sound pointer passed to it with a stream.
- **Studio API** - Fixed incorrect parameter values being passed to nested events when value "hold" is being used.
- **Studio API** - **PS3** - Fix potential crash with the new Channel Mix effect.
- **FSBank API** - Fixed encoder bug with FADPCM causing occasional clipping at playback.

Notes:

- **FSBank API** - FSbs / Banks may not be binary identical to previous release due to FADPCM encoder bug fix, however full compatibility is maintained.
20/10/16 1.08.14 - Studio API minor release (build 80900)

Fixes:

- **UE4** - Fix for crash when using "Validate FMOD" menu item

Notes:

- **PS4** - Now built with SDK 4.00
- **Xbox One** - Added better logging and documentation to describe an incorrectly configured appxmanifest regarding microphone recording.
04/10/16 1.08.13 - Studio API minor release (build 80479)

Fixes:

- **Studio API** - Fixed potential crash after the following sequence of actions: load master bank, try to load master bank again and fail with `FMOD_ERR_EVENT_ALREADY_LOADED`, unload master bank, reload master bank.
- **LowLevel API** - Fix circular DSP connection causing hang in certain situations.
- **Unity 2** - Fix issues with multi-object editing of emitters.
22/09/16 1.08.12 - Studio API minor release (build 80229)

Features:

- *Unity 2* - Added ability to override minimum and maximum distance for Event emitters.
- *Unity 2* - Added support for multiple listeners.
- *Studio API* - Added support for auto pitch at minimum.
- *Studio API* - Added support for the global master bus being duplicated across banks.

Fixes:

- *Studio API* - Fix auto pitch cutting off at zero for parameter with minimum value that is less than zero.
- *Studio API* - Fix events with a transciever effect not allowing the event to stop
- *Android* - Fixed crash when loading FSBs or Banks that contain a sound that isn't mono, stereo, 5.1 or 7.1.
- *Android* - Fixed compatibility issue with some devices introduced in previous release due to r12b update. Presents as a runtime linker error when loading the FMOD library, failing to locate __aeabi_atexit.
- *iOS* - Fixed stuttering during fade out when device screen goes to sleep.
- *LowLevel API* - Fix `FMOD_DSP_TRANSCEIVER` memory stomp.
- *LowLevel API* - Fix channels playing at incorrect pitch. Introduced in 1.08.10.

Notes:

- *Studio API* - Incremented bank version, requires runtime 1.08.00 or later.
- *Studio API* - Latest runtime supports loading old bank versions from 1.03.00.
- *LowLevel API* - Improved validation for ChannelGroup, DSP and Sound handles, detects invalid pointers and usage after release.
08/09/16 1.08.11 - Studio API minor release (build 79819)

Features:

- **LowLevel API - PS3** - Added support for `FMOD_DSP_CHANNELMIX`.
- **Unity 2** - Respect Game View mute button.

Fixes:

- **FSBank** - Fix crash on 64-bit when encoding 16kHz or 24kHz sources using Vorbis at low quality settings.
- **LowLevel API** - Fix `FMOD_SOUND_PCMSETPOS_CALLBACK` getting invalid position value when a sound opened with `FMOD_OPENUSER` loops.
- **LowLevel API - Android** - Fixed crash on load with old devices when using armeabi.
- **Unity 2** - Fix `CREATESOUNDEXINFO` not getting marshalled properly.

Notes:

- **Android** - Now built with NDK r12b.
- **Android** - Minimum Android version is now API level 9 due to NDK r12b deprecating older versions.
22/08/16 1.08.10 - Studio API minor release (build 79252)

Features:

- *Studio API* - Improved performance of sidechain modulator.
- *LowLevel API* - Improved `ChannelControl::setPitch` accuracy between DSP clock and the underlying codec decoding speed.
- *Unity 2* - Added option for play-in-editor to reflect the active build target for loading banks.

Fixes:

- *LowLevel API* - Fixed error trying to create a Return DSP when the system format is set to `FMOD_SPEAKERMODE_RAW`.
- *LowLevel API* - Fixed FFT DSP crash if window size is smaller than DSP block size.
- *LowLevel API* - Removed spurious warning messages when loading plugins on some platforms.
- *LowLevel API - Android* - Tweaked OpenSL auto detection, now requires device to specify low latency and a block size <= 1024.
- *Unity 2* - Fix bank import issues when the strings bank contains bank names that differ in case from the files on disk.
- *Unity 2* - Bank import now has a longer timeout after last detected file activity before starting import.
- *Unity 2* - Fixed settings screen allowing real channels to be set higher then 256.
- *Unity 2* - Fix up errors when StudioEventEmitter is created dynamically.
- *Unity 2* - Small fixes for the settings screen when overriding the parent platform settings.
- *UE4* - Fix for crash when previewing animations using the FMOD event notifier.

Notes:

- *LowLevel API* - `System::getSpeakerModeChannels` returns the system
channel count when passed \texttt{FMOD\_SPEAKERMODE\_DEFAULT}. Now works even if the system format is set to \texttt{FMOD\_SPEAKERMODE\_RAW}.

- \textit{UE4} - Can be recompiled with 4.13 pre-release.
- \textit{Documentation} - Added documentation for \texttt{FMOD\_DSP\_PAN} enums.
01/08/16 1.08.09 - Studio API minor release (build 78489)

Features:

- **LowLevel API - PS4** - Add support for social screen audio to the ports API.
- **Studio API** - Added Studio::EventInstance::setListenerMask and Studio::EventInstance::getListenerMask, that adds the ability to specify which listeners apply to each event instance.
- **Unity 2** - Warning is now produced when playing in editor if the position of a 3D event is not set.
- **UE4** - Added blueprint functions to set event properties.

Fixes:

- **Studio API** - Fixed case of indeterminism when building banks that contain events with automation curves.
- **LowLevel API** - Fixed FMOD_OUTPUT_OBJECT3DINFO::gain so it only includes distance attenuation not bus gain (which is now pre-applied to FMOD_OUTPUT_OBJECT3DINFO::buffer.
- **LowLevel API** - Fixed channelmix DSP volume not always being initialized.
- **LowLevel API - WiiU** - Fixed potential crash during System::init if System::setDriver or System::setOutput has been called.
- **LowLevel API** - Fix hang on netstreams when the connection times out.
- **LowLevel API - Linux** - Fix FPU control word of the calling thread being modified when the FMOD dynamic library is loaded.
- **LowLevel API - Android** - Fixed potential crash if FMOD isn't loaded with System.loadLibrary, now a proper error will be issued.
- **FSBank API** - Fixed FADPCM not looping seamlessly for non-zero crossing loops.
- **UE4** - Fixed plugin loading assuming a "lib" prefix for plugins on Android, Mac, Linux and PS4. Now plugin loading will attempt to load the name with and without adding a lib prefix.
- **UE4** - Respect FApp::IsUnattended for message-box errors.
- **UE4** - Fixed deprecation warnings about AttachTo usage.
• *Unity 2* - Fix errors when bank source files are updated while Event Browser preview is playing or paused.
• *Unity 2* - Fix unnecessary copying of Bank files in OSX editor.

**Notes:**

• *LowLevel API - Linux* - Removed limit of 32 devices with ALSA output mode.
• *LowLevel API* - Incremented API version of Output plugins. Dynamic library plugins built with earlier versions of 1.08 will continue to load.
Features:

- **UE4** - bMatchHardwareSampleRate will use the system default format to avoid excessively matching the output rate on mobile devices.
- **UE4** - Added bLockAllBuses which will force all buses to be created at startup, rather than on demand.

Fixes:

- **Studio API** - Fixed looping sounds in a multi sound playlist failing to stop if the multi sound itself is non-looping.
- **Studio API** - Fixed rare crash calling the following functions while the bank containing the event is unloading: `Studio::EventInstance::getDescription`, `Studio::EventInstance::getParameter`, `Studio::EventInstance::getParameterValue`, `Studio::EventInstance::setParameterValue`, `Studio::EventInstance::setParameterValueByIndex`, `Studio::EventInstance::triggerCue`, `Studio::ParameterInstance::getDescription` and `Studio::ParameterInstance::setValue`.
- **Studio API** - Fixed shared events becoming invalidated when one of the banks containing them is unloaded. Could also manifest as `Studio::EventInstance::getDescription` failing with `FMOD_ERR_INTERNAL`.
- **Studio API** - Fixed crash that could occur when using `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE` and calling `Studio::System::update` from multiple threads at the same time.
- **Studio API** - Increased scheduling lookahead time to ensure sample accurate timeline scheduling even if there is an output stall.
- **Studio API** - Fixed crash that could occur when improperly calling `Studio::Bus::getChannelGroup` without first calling `Studio::Bus::lockChannelGroup`, if the bus was being destroyed as the call was made.
• **Studio API** - Fixed rare crash calling the following functions while the master bank containing the bus or VCA is unloading:
  Studio::System::getBus, Studio::System::getBusByID, Studio::System::getVCA and Studio::System::getVCAByID.

• **Studio API** - Fixed rare timing issue where Studio::System::getEvent would succeed but Studio::EventDescription::createInstance would fail with FMOD_ERR_INTERNAL, if called just as the bank containing the event finished loading.

• **Studio API** - Fixed rare hang in Studio::System::getBankCount when called while banks are currently unloading.

• **LowLevel API** - Fixed rare glitch at the start of XMA playback causing non-seamless looping.

• **LowLevel API** - Fixed rare hang on shutdown when using multiple systems.

• **LowLevel API** - Fix streams with an unknown file length remaining in the playing state after an end of file is encountered.

• **LowLevel API - Windows** - Enumeration of record devices will now reflect a change of default device with WASAPI output.

• **LowLevel API - Linux** - Enumeration will now correctly display GUIDs and speaker modes for ALSA output.

• **LowLevel API - Linux** - Fixed potential crash if both PulseAudio and ALSA are missing / unavailable.

• **Unity 2** - Fix plugin loading on Linux standalone builds.

• **Unity 2** - Fix script compilation errors in standalone builds.

• **Unity 2** - Fix Event Browser preview of events with built-in parameters.

• **Unity 2** - Fix missing tvOS files.

**Notes:**

• **LowLevel API** - Significantly reduced memory consumption when using FMOD_DSP_FFT.

• **PS4** - Now built with SDK 3.508.101

• **UE4** - Updated Oculus plugin to 1.0.4.
27/06/16 1.08.07 - Studio API minor release (build 77241)

Features:

- **FSBank API** - Added support for Linux.
- **Unity 2** - Live Update and Debug Overlay can be set to only be enabled in standalone builds when the development build option is set.
- **Unity 2** - Add MuteAllEvents and PauseAllEvents functions to the RuntimeManager.
- **Unity 2** - Audio will now pause when the application pauses.
- **UE4** - Added MixerSuspend and MixerResume blueprint functions.
- **UE4** - Added IsBankLoaded blueprint function.

Fixes:

- **Studio API** - Fixed Studio command buffer assert and crash that could occur when using `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE` in conjunction with multiple threads.
- **LowLevel API** - Fix crash when closing a system with a multi-plugin DLL still loaded.
- **LowLevel API - iOS / Android** - Improved audio stability when mixer thread is overloaded.
- **FSBank API** - Fixed calling convention linker error on Windows in FSBankLib header.
- **FSBank API** - Fixed issue when passing duplicate source files with different encoding settings would cause cache file conflicts.

Notes:

- **Unity 2** - Renamed "Level Start" and "Level End" triggers to "Object Start" and "Object Destroy" to more accurately reflect when the triggers occur.
17/06/16 1.08.06 - Studio API minor release (build 76937)

Features:

- *Unity 2* - Added framework for registering native plugins on iOS and tvOS.
- *Unity 2* - Added support for importing Studio Banks as TextAssets so they can be added to Unity AssetBundles.

Fixes:

- *LowLevel API* - Fixed crash on shutdown, after creating multiple systems and setting a system callback.
- *Lowlevel API - PS4* - Fix issues with calling FMOD_Orbis_GetPadVolume() immediately after opening the controller audio port.
Features:

- **Studio API** - Added `Studio::EventDescription::isSnapshot`.
- **Unity 2** - Added "Preload Sample Data" checkbox to Event Emitter to reduce latency when emitters are triggered for the first time.
- **Unity 2** - Added script example to show how to build an asynchronous loading screen that includes loading FMOD Banks.

Fixes:

- **Studio API** - Fixed `Studio::EventDescription::isOneshot()` incorrectly returning true for snapshots.
- **LowLevel API** - Fix FMOD_ADVANCEDSETTINGS::maxADPCMCODECS not being applied.
- **LowLevel API** - Fix crash when using CreateSound from multiple threads (including FMOD Async loading threads).
- **Unity 2** - Fix issues when bank file in the Unity project Streaming Assets folder have a different case to the banks in the Studio project.
- **Unity 2** - Fix issues when editor log file cannot be opened because it's read only.
- **Unity 2** - If the "Load All" options are selected in the FMOD Settings then the main thread will now block until it's complete.
- **UE4** - Fix for OnEventStopped callback firing repeatedly if triggers the instance to play again.

Notes:

- **LowLevel API** - All System, ChannelControl, ChannelGroup, Channel, and DSP API functions check for NaN and INF floats and return `FMOD_ERR_INVALID_FLOAT` if detected.
- **Studio API** - All API functions check for NaN and INF floats and return `FMOD_ERR_INVALID_FLOAT` if detected.
- **Studio API** - All API functions with output parameters now clear those
values in the case of an error. Previously some values may have been left uninitialized. In the case of an error, int and float outputs are set to 0, bool outputs are set to false, and pointers are set to NULL. Structures are cleared to zeros, and string buffers are set to the empty string.
25/05/16 1.08.04 - Studio API minor release (build 76196)

Features:

- *Studio API* - Added runtime support for steal quietest polyphony.
- *LowLevel API* - Improved performance when connecting sends and returns.
- *UE4* - FMODEventInstance can now be stored as a blueprint variable.
- *UE4* - Added support for Occlusion. See the Occlusion section of the documentation for more information.
- *UE4* - Added support for Android deployment without having to modify the engine.
- *UE4* - Added InitialDriverOutputName to select output device by name at startup, as well as new Blueprint functions GetOutputDrivers, SetOutputDriverByName and SetOutputDriverByIndex.
- *UE4* - Added VCASetFaderLevel Blueprint function.
- *UE4* - "FMOD Validate" now checks for FMOD in the packaging setting, and can add it if needed.

Fixes:

- *LowLevel API* - Fixed unnecessary DSP queue flush when using ports.
- *LowLevel API* - Fixed ADPCM and FADPCM compressed FSBs not returning `FMOD_ERR_MEMORY_CANTPOINT` when loaded as `FMOD_OPENMEMORY_POINT FMOD_CREATERSAMPLE`.
- *LowLevel API* - Fix pops when channels with a mix matrix that are started virtual become real.
- *LowLevel API* - Fixed DSP panner reset not clearing ramps, causing ramping when setting initial parameters.
- *LowLevel API* - Fixed `Sound::getSyncPointInfo` returning first sync point info when loading sounds from FSBs with multiple sync points.
- *Studio API* - Fixed memory stomp that can occur when sharing events between multiple banks, if a streaming sound is in the middle of loading when one of the shared banks is unloaded.
- *Unity 2* - Fix error messages when previewing an event contained in the Master Bank.
• *Unity 2* - Fix WinPhone 8.1 DLL's conflicting with UWP builds.

**Notes:**

• *Studio API* - Incremented bank version, requires runtime 1.08.00 or later.
• *Studio API* - Latest runtime supports loading old bank versions from 1.03.00.
• *UE4* - Updated ovrfmod to version 1.0.3.
• *UE4* - Tested with 4.12 pre-release, compiles successfully.
• *PS Vita* - Now built with SDK 3.570.011.
Features:

- Studio API - Added
  FMOD_STUDIO_EVENT_CALLBACK_SOUND_PLAYED  and
  FMOD_STUDIO_EVENT_CALLBACK_SOUND_STOPPED  for when an
  event plays sounds. Sound names will only be available if banks have been
  re-exported in FMOD Studio 1.08.03 or later. See the music_callbacks
  example for demonstration.

- Studio API - Added runtime support for the event cooldown property set
  from FMOD Studio. Events that fail to start due to cooldown time will
  invoke the  FMOD_STUDIO_EVENT_CALLBACK_START_FAILED
  callback.

- LowLevel API - Improved performance of logging.

- LowLevel API - PS4 - Improved performance when profiling is enabled.

Fixes:

- Studio API - Events that fail to start due to bus polyphony now invoke the
  FMOD_STUDIO_EVENT_CALLBACK_START_FAILED callback.

- Studio API - Fixed crash when calling Studio::System::unloadAll with
  crossfading nested events.

- Studio API - Fixed unnecessary up/down mix applied to 2D events that
  have sidechain modulators.

- Studio API - Fixed mixing issues and degraded performance if the system
  speaker mode does not match the Studio project format.

- Studio API - Fixed case of stereo sounds panning hard right when
  downmixed to a mono track and later upmixed again, introduced in 1.07.04.

- LowLevel API - Fixed potential incorrect position when voice comes back
  from being virtual, which can cause a hang on XboxOne.

- LowLevel API - Improved handling for out of memory errors when mixing
  DSP buffers.

- LowLevel API - Fixed incorrect propagation of
  FMOD_DSP_STATE::channelmask when mixing signals with differing
masks.

- **LowLevel API** - Fixed http streams (file from http, not shoutcast/icecast) returning `FMOD_ERR_FORMAT`. Introduced 1.08.01
- **LowLevel API** - Fixed m3u playlist file format support. Was returning `FMOD_ERR_FORMAT`.
- **LowLevel API** - **Android** - Improved detection of low-latency devices allowing better automatic output mode selection and less stuttering.
- **LowLevel API** - **Sound::getName** will now return "" for sounds without names, instead of "(null)".
- **LowLevel API** - Object 3D Panning fix for silent objects in certain speaker modes
- **Studio API** - More robust live update handshaking when attempting to connect with multiple copies of FMOD Studio at once.
- **Unity 2** - Added missing **Studio::EventDescription::getSoundSize** function.

**Notes:**

- **Studio API** - Incremented bank version, requires runtime 1.08.00 or later.
- **Studio API** - Latest runtime supports loading old bank versions from 1.03.00.
- **Studio API** - Sound names are now loaded into memory if they have been exported in the bank file. The option is on by default, which means by the runtime memory use might be slightly higher when loading banks exported from 1.08.03 and later. If this is a problem, make sure to disable the option to export sound names in FMOD Studio when re-exporting banks.
- **Xbox One** - Now built with March 2016 QFE 1 XDK.
- **PS4** - Now built with SDK 3.508.031.
13/04/16 1.08.02 - Studio API minor release (build 74770)

 Fixes:

- **UE4** - Fixed audio component transform not being updated in 4.11.
- **Studio API** - Fixed unnecessary up/down mix applied to 2D events that have sidechain modulators.
- **Studio API** - `Studio::EventDescription::is3D` now returns true if there is a plug-in panner on the master track.

 Notes:

- **UE4 - PS4** - Deployment uses standard unreal plugin system.
- **UE4** - Now built against Unreal 4.11
07/04/16 1.08.01 - Studio API minor release (build 74554)

Features:

- *Unity 2* - Added Universal Windows Application platform support.
- *Unity 2* - Added Find and Replace tool.
- *LowLevel API* - Improved performance when creating DSPs.
- *UE4* - Added FMOD stats for CPU usage.

Fixes:

- *Studio API* - Fixed errors being generated when the initial seek is past the loop end point of a sound.
- *Studio API* - Fixed error loading sample data for banks loaded by `Studio::System::loadBankMemory` with `FMOD_STUDIO_LOAD_BANK_DECOMPRESS_SAMPLES` flag.
- *Studio API* - Fixed rare Studio update error `FMOD_ERR_NOTREADY` when stopping modules with streaming sounds.
- *Studio API* - Fixed unnecessary up/down mix on sidechain effects in game.
- *Studio API* - Eliminated API stalls due to holding a lock when creating event instances on the Studio Update thread.
- *Studio API* - Fixed error when loading an API capture that contains `Studio::System::flushSampleLoading` commands.
- *Studio API - PS4* - Fixed incorrect linking options on fmodstudio.prx that caused package creation to fail.
- *LowLevel API* - Fixed rare hang when rapidly creating and releasing Return DSPs.
- *LowLevel API* - Fixed hang or crash when loading a .it/s3m/mod/mid file as a decompressed sound.
- *LowLevel API* - Fixed some shoutcast streams playing back garbled.
- *LowLevel API* - Fixed `Sound::readData` returning 0 for bytes read, instead of a valid number if `FMOD_ERR_FILE_EOF` was returned. Introduced in 1.07.08.
- *LowLevel API - PS4* - Fix AT9 playback when a sound created as looping is played with the channel loop-count explicitly set to zero before starting.
- **LowLevel API - Win** - Fixed ASIO device enumeration not supplying a GUID.
- **LowLevel API - Win** - Fixed ASIO device enumeration having a blank name if the device is not connected.
- **UE4** - Added lock around file accesses to avoid Unreal pak file thread safety issue.
- **UE4** - Fixed logging callback not being initialized.
- **UE4** - Avoid asset table system from mixing while normal mixer is in operation, to work around an AT9 mixer issue.
- **UE4** - Fixed always linking against ovrfmod even if it isn't present.
- **Unity 2** - Rewrote Timeline Callback and Programmer Sound Callback examples to work on iOS.
- **Unity 2** - Fix marshalling of FMOD.CREATESOUNDEXINFO structure on iOS.
- **Unity 2** - Fix DLL not found errors in standalone Windows builds.

**Notes:**

- **Studio API** - Incremented bank version, requires runtime 1.08.00 or later.
- **Studio API** - Latest runtime supports loading old bank versions from 1.03.00.
- **Studio API** - [Studio::EventInstance::setPitch](#) now returns an error if a NaN is passed in.
- **Studio API** - Errors that occur during the Studio update thread will no longer stop the thread.
- **Studio API** - Studio will set the low level master channel group format to the project's format to avoid an extra upmix.
Features:

- **Studio API** - Sample data loading has been optimised. Load time, file access, and memory use have all been substantially improved. A new entry has been added to the per platform thread affinity settings.
- **Studio API** - `FMOD_STUDIO_PARAMETER_DESCRIPTION` now has the parameter index and default value.
- **Studio API** - Added `Studio::System::flushSampleLoading`.
- **Studio API** - Support for left edge trimming of timelocked sounds.
- **Studio API** - Support for Start Offset as a percentage of sound length.
- **Studio API** - Added idle resource pool to keep recently used sounds in memory in case they might be re-used. It can be controlled by the `idleResourcePoolSize` field in `FMOD_STUDIO_ADVANCEDSETTINGS`. See the Studio Banks Programmer Topic for more information.
- **LowLevel API** - Increased performance of `System::createSound` and `System::createStream`, since they no longer block against `System::update`.
- **LowLevel API** - Added filebuffersize to `FMOD_CREATESOUNDEXINFO` for customizable file buffering.
- **LowLevel API** - Added `System::getFileUsage` to query file loading information.
- **LowLevel API** - Custom DSP effects now always receive a buffer length that is equal to the mix block size. The input and output buffers will always be 16-byte aligned. Custom DSP sounds still have be able to generate signal less than a mix block.
- **LowLevel API** - Added getclock callback to `FMOD_DSP_STATE_SYSTEMCALLBACKS` to get the clock, offset and length for a custom DSP.
- **LowLevel API** - Added support for multiple plugins within one plugin file. See `FMOD_PLUGINLIST`, `System::getNumNestedPlugins`, `System::getNestedPlugin`, and the DSP Plugin API Programmer Topic for more information.
- **LowLevel API** - Added support for object based panning with two backend providers, Dolby Atmos (`FMOD_OUTPUTTYPE_ATMOS`) and Playstation VR (`FMOD_OUTPUTTYPE_AUDIO3D`).
- **LowLevel API** - Added 3D object panner DSP (`FMOD_DSP_TYPE_OBJECTPAN`) to be used with new object pan enabled outputs.
- **LowLevel API** - Extended output mode plugin API (`FMOD_OUTPUT_DESCRIPTION`) to allow custom object panner backends.
- **LowLevel API** - Win - Reduced WASAPI latency by 40ms and improved mixer thread regularity.
- **FSBank** - AT9 Band Extension is now enabled by default for supported bit-rates.
- **FSBank** - Added Mac versions of the FSBank tool and FSBankLib API. Included in the Mac and iOS API packages.
- **Profiler** - Added Mac version of the Profiler tool. Included in the Mac and iOS API packages.
- **Unity 2** - Added ability to create new studio events from within the Unity editor.
- **Unity 2** - Improved event selection UI on emitter component and when using EventRef attribute.
- **Unity 2** - Added support for default parameter values in emitter component.

**Notes:**

- **Studio API** - Incremented bank version, requires runtime 1.08.00 or later.
- **Studio API** - Latest runtime supports loading old bank versions from 1.03.00.
- **Studio API** - New example 'object_pan' that demonstrates object based panning.
- **Studio API** - EventDescription::getSampleLoadingState and Bank::getSampleLoadingState now return `FMOD_STUDIO_LOADING_STATE_ERROR` if sample data failed to load (e.g. due to a corrupt file).
- **Studio API** - Removed Studio::CueInstance, Studio::EventInstance::getCue, Studio::EventInstance::getCueCount and Studio::EventInstance::getCueByIndex. Instead use new functions Studio::EventDescription::hasCue and Studio::EventInstance::triggerCue.
- **Studio API** - Deprecated Studio::EventInstance::getParameter and Studio::EventInstance::getParameterByIndex. Instead use Studio::EventInstance::getParameterValue, Studio::EventInstance::setParameterValue,
Studio::EventInstance::getParameterValueByIndex, and Studio::EventInstance::setParameterValueByIndex.

- **Studio API** - Increased stack size for Studio threads to 64K.
- **Studio API** - Played events will stay in the **FMOD_STUDIO_PLAYBACK_STARTING** state until their sample data has loaded. This avoids selected sounds in the event playing late if the sample data has not been preloaded.
- **LowLevel API** - System::getChannelsPlaying now returns the number of real and total channels playing. System::getChannelsReal has been removed.
- **FSBank** - AT9 compression now requires AT9 library 1.7.1 (DLL version 2.8.0.5) or later. Compression in 32bit versions of FSBank is no longer supported in line with Sony's removal of 32bit compression libraries.
03/03/16 1.07.08 - Studio API patch release (build 73591)

Features:

- *Unity 2* - Importing banks has been speed up dramatically.

Fixes:

- *Studio API* - Fixed *Studio::EventInstance::setProperty* not restoring the default setting for
  
  - `FMOD_STUDIO_EVENT_PROPERTY_MINIMUM_DISTANCE`
  - `FMOD_STUDIO_EVENT_PROPERTY_MAXIMUM_DISTANCE`

  when a value of -1 is specified.

- *Studio API* - Fixed case of mono sounds panning hard left when both mono and stereo 2D sounds are placed on the same event track, introduced in 1.07.04.

- *LowLevel API* - Fixed some net streams returning 'invalid parameter' introduced in 1.07.06.

- *LowLevel API* - Fixed potential crash if calling *Sound::release* soon after
  
  - *System::createSound* with `FMOD_NONBLOCKING` and async IO callbacks.

- *LowLevel API - Winstore/UWP* - Fixed small memory leak on system shutdown.

- *LowLevel API - Winstore/UWP* - Fixed occasional crash when closing a socket with a pending asynchronous read.

- *LowLevel API - Xbox One* - Fixed potential hang waiting on XMA *Sound::release* if using async IO callbacks and a cancel was issued.

- *LowLevel API - Android* - Relaxed overly strict validation of M4A files introduced in 1.07.07.


- *LowLevel API* - Fix crash in convolution reverb if wet level is set to -80db and it has no inputs.

- *LowLevel API* - Fix seeking on stereo FADPCM compressed streams and samples.
• *Unity 2* - Errors opening log output file are no longer fatal.

**Notes:**

• *LowLevel API* - Ensure FMOD_FILEASYNCDONE is called with
  [FMOD_ERR_FILE_DISKEJECTED](https://www.fmod.org/files/FMODAPI/API/FMOD_ERR_FILE_DISKEJECTED) if implementing
  [FMOD_FILEASYNCANCEL_CALLBACK](https://www.fmod.org/files/FMODAPI/API/FMOD_FILEASYNCANCEL_CALLBACK) to notify FMOD that you
  will not be servicing the FMOD_ASYNCREADINFO::buffer.
16/02/16 1.07.07 - Studio API patch release (build 72710)

Features:

- *LowLevel API - Win* - Improved ASIO output to accept requested sample rate and buffer size instead of using defaults.
- *UE4* - FMOD Memory allocation now occurs via standard UE4 allocators.
- *Unity 2* - Added AppleTV support.

Fixes:

- *LowLevel API* - Fixed `System::loadPlugin` not unloading the library file in the case of an error.
- *LowLevel API* - Fixed automatic format detection failing for some types when the file size is less than 1KB.
- *LowLevel API* - Fixed `FMOD_CREATESOUNDEXINFO::suggestedsoundtype` being ignored for `FMOD_OPENMEMORY` and `FMOD_OPENMEMORY_POINT`.
- *LowLevel API - WinStore/UWP* - Fixed occasional deadlock when removing the current output device.
- *LowLevel API - Win* - Fixed potential crash if switching output mode to ASIO after `System::init`.
- *LowLevel API - Android* - Fixed crash if MediaCodec processes a file that isn't an M4A file.
- *LowLevel API - Android* - Fixed Marshmallow devices being unable to load M4A files.
- *Unity 2* - Remove broken inspector code for ParamRef.
- *Unity 2* - Fix PS Vita.
- *Unity 2* - Small event reference UI rendering fixes.
- *Unity 2* - Fix unnecessary copying of files that haven't been modified.
- *Unity 2* - Fix issues with copying of files that have only been partially written by the Studio tool.
- *Unity 2* - Work around IL2CPP issues introduced in Unity 5.3.2p2.
- *Legacy Unity* - Work around IL2CPP issues introduced in Unity 5.3.2p2.
- *Unity 2 - Xbox One* - Fix runtime errors when using Mono AOT.
compilation.

Notes:

- *Unity 2* - Unity 5.3.2p1 is not supported for iOS.
- *Legacy Unity* - Unity 5.3.2p1 is not supported for iOS.
27/01/16 1.07.06 - Studio API patch release (build 71893)

Features:

- Studio API - Improved CPU and IO performance when capturing API commands via Studio profiler.
- LowLevel API - Android - Lollipop devices may now select 7.1 with AudioTrack output mode. Older devices will gracefully fall back to stereo.
- Unity 2 - Emitter gizmos are now pickable in the scene view.
- Unity 2 - Event Browser layout changed to work in narrow windows. Note the Event Browser windows will need to be closed and re-opened for new minimum bounds to take effect.
- Unity 2 - Added bitcode support for iOS.
- UE4 - Added support for programmer sounds. See the new UE4 Programmer Sound page for more information.

Fixes:

- LowLevel API - Fixed bug if streams start virtual, they may not come back as audible. Introduced in 1.07.00
- LowLevel API - Update URL in Netstream example.
- LowLevel API - Fixed net streams not working when server delivers shoutcast with chunked HTTP data.
- LowLevel API - Fixed memory leak when a network connection fails.
- LowLevel API - Android - Devices that don't support 5.1 output will now gracefully fallback to stereo.
- LowLevel API - WinStore / UWP - Fixed WASAPI output so it return the correct 'no drivers' error when none are detected.
- LowLevel API - WinStore / UWP - Fixed WASAPI output swapped sides and rears in 7.1.
- LowLevel API - Fix C# wrapper of ChannelGroup.addGroup missing arguments.
- LowLevel API - Fix crash when using chorus effect on a channel group with channel count greater than the system channel count.
• **LowLevel API - PS4** - Fix validation of thread affinity settings.
• **LowLevel API - iOS** - Fixed compatibility issue introduced in 1.07.02 for old devices running iOS 6.X.
• **Unity 2** - Fixed UI when EventRef attribute is used on an array of strings.
• **Unity 2** - Work around DLL loading issues on some standalone Windows builds.
• **Unity 2** - Fix DLL loading issues for play-in-editor when platform is iOS.
• **Unity 2** - Fix RuntimeManager being recreated during shutdown and leaking FMOD native instances.
• **Unity 2** - Fix issue when using RuntimeManager.LowLevelSystem and RuntimeManager.StudioSystem before RuntimeManager has initialized.
• **Unity 2** - Increase channels for Play-In-Editor.
• **Unity 2** - Fix play-in-editor not setting the speaker mode correctly.
• **Unity 2 - PS4** - Fix script compile errors.

**Notes:**

• **Xbox One** - Now built with November 2015 XDK.
• **iOS** - Now built with iOS SDK 9.2 and tvOS SDK 9.1 (Xcode 7.2).
• **UE4** - Updated Oculus plugin for 1.0.1.
07/01/15 1.07.05 - Studio API patch release (build 71238)

Features:

- *Unity 2* - When using the file browser in the settings editor to set the Studio source paths the result will be set relative to the Unity project if it is a sub-directory.
- *Unity 2* - Added Parameter Trigger component.

Fixes:

- *LowLevel API - UWP* - Fixed `FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED` not firing when the default output device changes.
- *LowLevel API - PS3* - Fix rare crash with reverb.
- *LowLevel API - PS3* - Fix static and audio dropouts if speaker mode is forced to stereo (normally it is 7.1).

Notes:

- *iOS* - Reduced binary size.
- *Android* - Added calling convention to public API to allow usage with hard float ABI.
11/12/15 1.07.04 - Studio API patch release (build 70728)

Features:

- **LowLevel API** - Better support for sounds with channel/speaker masks, ie a 1 channel sound specified as LFE, or a quad sound specified as LRCS instead of 2 front 2 back for example.
- **Unity 2** - Added optional debug overlay.
- **Unity 2** - Added migration steps for data in custom components.
- **Unity 2** - Added support for Unity 4.6.
- **Unity 2** - Added support for Android split binary.
- **Unity Legacy** - Added support for Android split binary.

Fixes:

- **Studio API** - Fixed blocking bank loads stalling for longer than necessary when other threads are constantly issuing new Studio commands.
- **LowLevel API** - Fixed corrupted playback for big endian PCM FSBs.
- **LowLevel API** - Fixed crash when switching channel count of generator DSPs during playback.
- **LowLevel API - PS3** - Fix crash reallocating buffers on flange effect.
- **LowLevel API - Xbox One** - Fixed rare stall when playing XMA streams and compressed samples at the same time.
- **LowLevel API** - Fix crash if the Master Channel Group head DSP was set inactive right after initialization.
- **LowLevel API** - Fix Sound::getOpenState() moving from FMOD_OPENSTATE_PLAYING too early on some streaming sounds.
- **LowLevel API** - Fix Sound::Release() freeing memory still in use by the mixer.
- **LowLevel API** - Fix convolution reverb not correctly producing tails on long impulse responses when it has no input.
- **Unity 2** - Fixed issue with plugins not being loaded correctly.
- **Unity 2** - Work around DLL loading issues on some standalone Windows builds.
- **Unity 2** - Fix leaking system objects leading to ERR_MEMORY.
• *Unity 2* - Updated signature of `DSP::setWetDryMix`, `DSP::getWetDryMix`.

**Notes:**

• *LowLevel API* - `FMOD_DSP_STATE::channelmask` is now passed down through the dsp chain so plugin developers can see the original signal channel format even if a sound was upmixed.
17/11/15 1.07.03 - Studio API patch release (build 69975)

Features:

- **LowLevel API** - When using System::recordStart the provided FMOD::Sound can now be any channel count, up/down mixing will be performed as necessary.
- **LowLevel API** - Improved performance of convolution reverb effect when wet is 0 or input goes idle.
- **LowLevel API - PS4** - Added FMOD_THREAD_CORE6 to allow access to the newly unlocked 7th core.
- **LowLevel API - PS4** - Added FMOD_Orbis_GetPadVolume() to retrieve the output volume of the pad speaker as set by the user in the system software.
- **LowLevel API** - Added **FMOD_DSP_TYPE_TRANSCEIVER**. Send signals from multiple sources to a single 'channel' (out of 32 global 'channels') and receive from that or any channel from multiple receivers. Great for receiving and broadcasting a submix from multiple 3d locations (amongst other uses).
- **Unity** - FMOD_StudioSystem.GetEvent() can now be used with snapshot paths.
- **Unity** - Ignore OSX resource fork files when importing from FAT32 file systems.
- **UE4** - Deployment will also copy any plugins listed in a file plugins.txt in the FMODStudio/Binaries/Platform directory. See the Using Plugins page for more information.
- **UE4** - Console platforms set up thread affinity in FMODPlatformSystemSetup.

Fixes:

- **Studio API** - Fixed snapshot applied to VCA level not working in game.
- **Studio API** - Fixed profiler session timing when recording a game with a non-standard sample rate.
- **Studio API - PS4** - Fix linker error in examples.
- **LowLevel API** - Fixed buffer overrun when convolution reverb is set up
with mismatched input and output channels.

- **LowLevel API** - Fix crash with invalid .mp3 files.
- **LowLevel API** - Fix `Sound::getName()` not returning a valid UTF8 string when loading a sound with ID3 tags specifying the title as a latin1 string.
- **LowLevel API** - Fix pops in flange and chorus effects.
- **LowLevel API** - Fix crash when using flange effect on a channel group with channel count greater than the system channel count.
- **LowLevel API** - **FLAC** - Fix crash when seeking in certain flac files.
- **LowLevel API** - **Android** - Automatic selection of OpenSL output mode will now be stricter to reduce stuttering on devices that incorrectly report they are low latency.
- **LowLevel API** - **PS4** - Fix signal not being routed back to the main output when detaching a channel group from a port.
- **LowLevel API** - Fix FMOD_ADVANCEDSETTINGS::reverb3Dinstance not working.

**Notes:**

- **Android** - Now built with NDK r10e.
- **PS Vita** - Now built with SDK 3.550.
- **LowLevel API** - `FMOD_DSP_CHANNELGAIN_OUTPUT_DEFAULT ... FMOD_DSP_CHANNELGAIN_OUTPUT_ALLLFE` (from the `FMOD_DSP_CHANNELMIX_OUTPUT` enum) have been renamed to `FMOD_DSP_CHANNELMIX_OUTPUT_DEFAULT ... FMOD_DSP_CHANNELMIX_OUTPUT_ALLLFE` (ie CHANNELGAIN renamed to CHANNELMIX) to fit with the correct naming convention. Rename required if using this effect.
02/11/15 1.07.02 - Studio API patch release (build 69450)

Important:

- AppleTV platform support is now part of the iOS package.

Features:

- LowLevel API - Improved performance of System::getChannelsPlaying.
- LowLevel API - Added System::getChannelsReal to get the number of non-virtual playing channels.

Fixes:

- Studio API - Fixed internal error during playback caused by inaccuracies in quantization calculation when timeline position is greater than 5 minutes.
- LowLevel API - Fix mixer not running if mixer sample rate is lower than output sample rate.
- LowLevel API - iOS - Fixed output being mono on some devices, minimum detected hardware channel count is now 2 to ensure stereo.

Notes:

- iOS - Now built with iOS SDK 9.1 and tvOS SDK 9.0 (Xcode 7.1).
- Mac - Now built with SDK 10.11 (Xcode 7.1).
27/10/15 1.07.01 - Studio API patch release (build 69235)

Features:

- *Studio API* - Added support for multiple parameter conditions.
- *Studio API* - Added `Studio::EventDescription::getSoundSize`.
- *UE4* - Added validation help menu option to diagnose common issues.

Fixes:

- *Studio API* - Fixed auto-pitch, cutoff and looping not live updating properly.
- *LowLevel API* - Fixed potential crash on ARM platforms when seeking Vorbis compressed FSBs.
- *LowLevel API* - Fix `ChannelGroup::setReverbProperties` from returning `FMOD_ERR_REVERB_CHANNELGROUP` if a child channel had a reverb connection previously.
- *LowLevel API* - Fixed `FMOD_CREATESOUNDEXINFO::suggestedsoundtype` being ignored if a custom codec of higher priority has been registered.
- *LowLevel API* - Fixed `System::getRecordNumDrivers` incorrectly reporting 0 if called within 1 second of application start on some platforms.
- *LowLevel API* - Fixed stereo AIFF files producing static.
- *LowLevel API* - *Xbox One* - Fixed rare crash for stereo XMA streams.
- *LowLevel API* - *iOS* - Fixed MP3 decoding being performed by FMOD cross-platform decoder instead of native AudioQueue.
- *LowLevel API* - Fix mp3 crash seeking, introduced in 1.07.00 if using certain types of MP3 file in `FMOD_CREATECOMPRESSEDSAMPLE` mode.
- *Unity* - Change Android to read banks straight from APK
- *Unity* - Will automatically append a "64" suffix to plugins if required.
- *Unity* - Fix OSX bundles having incorrect settings
- *Unity* - Fix compatibility with Unity 5.2.2
- *Unity* - Fix crashes when creating a standalone OSX build.
- *UE4* - Fix for audio component direction not being set into FMOD.
- *UE4* - Fixed IOS deployment error "libfmod does not exist"
Notes:

- **Studio API** - The `FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_CREATED` and `FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_DESTROYED` callbacks now fire from nested events, for both plugin effects and plugin sounds.
- **Studio API** - For the `FMOD_STUDIO_PLUGIN_INSTANCE_PROPERTIES` callback argument, the 'name' field will contain just the user defined name of the plugin sound if one has been set. Otherwise it will be empty. You can use the 'dsp' field to determine the plugin type.
- **Studio API** - Programmer sounds with no name will have an empty string for `FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES.name`, previously it was an arbitrary hex sequence.
- **LowLevel API** - **Plugin developers** - `FMOD_DSP_SYSTEM_GETSPEAKERMODE` added to `FMOD_DSP_STATE_SYSTEMCALLBACKS`. Plugin SDK version updated to 1.07 meaning plugins created from this point onwards wont work with older versions of FMOD.
- **Android** - Replaced all Eclipse examples with Visual Studio using NVIDIA Nsight Tegra integration.
- **PS4** - Now built with SDK 3.008.041
- **PS3** - Now built with SDK 475.001.
05/10/15 1.07.00 - Studio API minor release (build 68517)

Important:

- Studio is now a 64-bit application. All plugin libraries must be built for 64-bit architecture to be compatible with the 64-bit version of the tool.
- Added Universal Windows Platform (UWP) support.
- Added AppleTV platform support. Contact support@fmod.org for beta access.

Features:

- **Studio API** - Support for bus polyphony.
- **Studio API** - Added `FMOD_STUDIO_EVENT_PROPERTY_MINIMUM_DISTANCE` and `FMOD_STUDIO_EVENT_PROPERTY_MAXIMUM_DISTANCE` to override the minimum and maximum panner distance for an instance.
- **Studio API** - Added `FMOD_STUDIO_INIT_DEFERRED_CALLBACKS` to defer event callbacks to the main thread. See `Studio::EventInstance::setCallback` for more information.
- **LowLevel API** - Added `FMOD_DSP_TYPE_CHANNELMIX` effect. Allows user to control gain levels for up to 32 input channel signals, and pipe the output to a range of speaker formats i.e. repeating mono, stereo, quad, 5.1, 7.1 or only LFE out.
- **LowLevel API** - Improved CPU performance when using metering or profiling.
- **LowLevel API** - Added support for loading multi-channel FSBs with the `FMOD_CREATECOMPRESSEDSAMPLE` flag.
- **LowLevel API** - Improved record driver enumeration, list is now persistent, removed devices will remain to ensure record IDs stay valid. See record_enumeration example for new functionality.
- **LowLevel API** - Added full record device add/removal detection for Windows, Mac, Xbox 360, Xbox One, PS3 and PS4.
- **LowLevel API** - Reduced recording latency on Windows, XboxOne and PS4.
- **LowLevel API - Mac** - Added support for recording from multiple microphones at the same time.
- **LowLevel API - Win** - Improved CPU performance for FSB Vorbis decoding. See Performance Reference section of the documentation for updated benchmark results.
- **LowLevel API - iOS** - Added support for bitcode.

**Fixes:**

- **Studio API** - Fixed the scatterer module's random distance calculation to ensure it is within the limit set in Studio. In previous versions the random distance may have exceeded the max limit. This change may affect the loudness of sounds with scatterer modules compared to 1.06.xx releases.
- **LowLevel API** - Fix rare crash when releasing a streamed sound.
- **LowLevel API** - Fix output plugins with GUIDs of zero stopping setDriver working.
- **LowLevel API - PS3** - Fixed send/returns not respecting volume changes or rarely failing with an error when setting return IDs.
- **LowLevel API - PS4** - Rewrote output to use sceAudioOutOutputs instead of multiple sceAudioOutOutput calls on a single thread.
- **LowLevel API - Win** - Fixed the audible output driver from resetting when an unrelated device is removed or disabled.
- **LowLevel API - Xbox One** - Fixed the first couple of samples of decoded XMA being lost due to SHAPE SRC.

**Notes:**

- **Studio API** - Incremented bank version, requires runtime 1.07.00 or later.
- **Studio API** - Latest runtime supports loading old bank versions from 1.03.00.
- **Studio API** - Improved compatibility for plugin effects. Adding extra DSP parameters to an effect no longer breaks bank version compatibility.
- **Studio API** - `Studio::Bank::getEventCount` and `Studio::Bank::getEventList` only return events directly added to the bank, not implicit events that are included in the bank due to event instrument references.
- **Studio API** - The asynchronous command buffer for the Studio API will now grow as necessary, avoiding stalls that could occur if it was too small. The `commandQueueSize` field in
**FMOD_STUDIO_ADVANCEDSETTINGS** now specifies the initial size of the buffer.

- **Studio API** - Studio now enables **FMOD_INIT_VOL0_BECOMES_VIRTUAL** by default.
- **Studio API** - **WinStore** - Fixed inconsistent naming of X64 libraries.
- **LowLevel API** - Increased **FMOD_MAX_LISTENERS** from 5 to 8.
- **LowLevel API** - **System::getCPUUsage** update time now includes the metering and profiling part of the update.
- **LowLevel API** - Removed support for FSB Vorbis files built using FMOD Ex 4.42 or earlier.
- **LowLevel API** - **PS4** - Changed default mix buffer size to 512 samples to reduce CPU cost. Previous setting of 256 can be set using **System::setDSPBufferSize**.
- **LowLevel API** - **PS4** - The size of each pre-allocated AT9 instance has increased by 8kB.
- **LowLevel API** - **PS4** - Prebuilt static libraries are no longer provided, only dynamic libraries.
- **LowLevel API** - **WinStore** - Fixed inconsistent naming of X64 libraries.
05/10/15 1.06.11 - Studio API patch release (build 68487)

Features:

- **Win** - Reduced dll and exe sizes.

Fixes:

- **Studio API** - Fixed input buses not being released when there are no longer any event instances routed into the bus.
- **Studio API** - Fixed API capture recording the initial VCA level as 0.
- **LowLevel API** - Fix rare crash when changing the master channel group head DSP when the software format and system output don't match.
- **LowLevel API** - Fix Channel::AddDSP() and ChannelGroup::AddDSP() not correctly moving a DSP if it's already a member.
- **LowLevel API** - Fix for noise being produced when panning a surround input when the system format doesn't match the panner input format.
- **Unity** - FMOD RNG seed set at initialization time.
- **Unity** - Can now properly cancel the copy step of the import process.
- **Unity** - Updated documentation links in menu.

Notes:

- **iOS** - Now built with SDK 9.0 (Xcode 7.0).
- **Mac** - Now built with SDK 10.11 (Xcode 7.0).
15/09/15 1.06.10 - Studio API patch release (build 67958)

Features:

- LowLevel API - Add `Sound::seekData` support to HTTP streams.
- UE4 - Added blueprint functions for loading and unloading sample data.

Fixes:

- LowLevel API - Fix small reverb pop on sounds that had `Channel::setPosition` called on then and were muted via a parent channelgroup.
- LowLevel API - Fix https netstream redirects not working.
- LowLevel API - Fix user DSP read callback firing with no data if `Channel::setPaused` (false) is used after a `System::playSound`.
- Studio API - Fixed Trigger Delay not being applied to sounds on parameters that trigger immediately when an event starts.

Notes:

- Xbox One - Now built with August 2015 XDK.
01/09/15 1.06.09 - Studio API patch release (build 67431)

Features:

- **Studio API** - Additional logging when banks have been exported with out of date plugin effects.

Fixes:

- **LowLevel API** - Fix automatic device changing support if a user called System::update from their own thread.
- **LowLevel API** - Fix crash when a channel group cannot be attached to port.
- **LowLevel API** - Fixed incorrect loop length being set for ADPCM files.
- **LowLevel API** - Win64 - Enable automatic device changing support.
- **LowLevel API** - iOS - Fixed error creating the file thread on iOS 9.0.
- **LowLevel API** - Fix distance filtering not resetting its parameters properly for channels.
- **LowLevel API** - MIDI support - Fix note keyoff when envelope is in decay phase causing note to finish early.
- **Studio API** - Fix crash that could occur when auditioning an event in Studio while changing the path to waveforms.

Notes:

- **UE4** - Updated oculus rift plugin to version 0.11
Features:

- **UE4** - Added FMOD init settings.
- **UE4** - Added preliminary support for 4.9 engine preview build.
- **LowLevel API** - Add `Channel::setPosition` support to HTTP streams.

Fixes:

- **LowLevel API** - Fixed `System::getDefaultMixMatrix` crashing if passed `FMOD_SPEAKERMODE_RAW`. Now returns `FMOD_ERR_INVALID_PARAM`.
- **LowLevel API - Xbox One** - Fixed crash if all SHAPE contexts are consumed.
- **LowLevel API - Xbox One** - Fixed XMA loops not being seamless.
- **Studio API** - Fixed potential crash when disconnecting from live update while recording a profiling session.
- **Studio API - WiiU** - Fixed hang on shutdown when exiting game from home menu.
- **LowLevel API** - Fix for inaccurate metering levels when the calculation spanned multiple mix blocks.
- **LowLevel API** - Fixed `System::getRecordPosition` race condition that could cause the returned value to be out of bounds.
- **Unity** - Fixed missing functionality from C# wrapper for beat callbacks and metering.
- **Unity** - Fix leaking native system objects in editor.
- **UE4** - Fixed multiple listeners.
- **FSBank** - Fixed crash encoding FADPCM format in 64bit version of tool.

Notes:

- **Studio API** - Reduced memory usage for instruments that do not require up/down mix.
- **LowLevel API** - Improved performance of `DSP::getParameterFloat`.
DSP::getParameterInt, DSP::getParameterBool and DSP::getParameterData when 'valuestr' argument is NULL.
22/07/15 1.06.07 - Studio API patch release (build 66161)

Fixes:

- *LowLevel API - Win* - Fix audio glitches at initialization time when using the WASAPI output mode on Windows 10.
- *LowLevel API* - Fix UTF8 strings in MP3 ID3V2 tags not being null terminated properly leading to possible string overflow crash.
- *Lowlevel API - PS4* - Fixed issue with non-1024 sample aligned loop-points and AT9 compressed samples.
- *Lowlevel API* - Fixed memory leak when releasing connections that have a user allocated mix matrix.
- *LowLevel API* - Added pre-wet and post-wet arguments to WetDryMix in C# wrapper.
- *Lowlevel API* - Fixed crash with FADPCM streams.
- *LowLevel API* - Fixed another potential crash in PitchShifter DSP if changing channel count while running.
- *LowLevel API - Win* - Fix issues caused by WASAPI allocating the output buffer at the mixer's sample rate instead of the device's sample rate.
- *Studio API - XboxOne* - Fix setting the thread affinity of the studio loading thread.
- *Studio API* - Fixed case of nested events on parameters being stopped while still active. Introduced in 1.06.04.

Notes:

- *Studio API* - Incremented bank version, requires runtime 1.06.06 or later.
- *Studio API* - Latest runtime supports loading old bank versions from 1.03.00.
- *LowLevel API - XboxOne* - Thread affinity can now be set as a mask of allowed cores.
08/07/15 1.06.06 - Studio API patch release (build 65638)

Features:

- **Lowlevel API** - Added [FMOD_DSP_COMPRESSOR_LINKED](#) parameter to DSP compressor.

Fixes:

- **Studio API** - Fixed potential invalid memory access when a bank is unloaded while create instance commands are queued.
- **Studio API** - Removed one frame delay between setting a parameter and having conditional timeline transitions occur.
- **Studio API** - Studio gracefully handles the case of a programmer sound being returned in a streaming not-ready state when the instrument is being used with timelocked seeks.
- **LowLevel API** - Fixed rare crash when streams go virtual.
- **LowLevel API** - Fixed 3EQ DSP not waiting an extra mix block before going idle, possibly cutting off a tail.
- **LowLevel API** - Fixed potential crash in PitchShifter DSP if changing channel count or FFT size while running.
- **LowLevel API** - Fixed rare volume pop when ramping with fade points as a channel goes emulated.
- **Lowlevel API - Linux** - Fixed record position not advancing with ALSA output mode.
- **LowLevel API** - Fix race condition when setting the impulse response on an active convolution reverb DSP.
- **Unity** - Fix compatibility with Unity 5.1
24/06/15 1.06.05 - Studio API patch release (build 65161)

Features:

- **Studio API** - Added `FMOD_STUDIO_LOAD_BANK_DECOMPRESS_SAMPLES` flag to force bank sample data to decompress into memory, saving CPU on low end platforms.
- **Studio API** - Improved responsiveness when reducing steal oldest polyphony over live update.
- **Studio API** - Studio profiling now includes nested instance information.
- **UE4** - Exposed beat and marker callbacks as Blueprint events.

Fixes:

- **Studio API** - Fixed Scatterer sounds having incorrect volume when min and max scatter distances are equal. Introduced in 1.06.00.
- **Studio API** - Fixed `FMOD_STUDIO_TIMELINE_BEAT_PROPERTIES` position to return the beat position, rather than the tempo marker position.
- **Studio API** - Fix deadlock in studio when removing an output device that runs at a different sample rate to the system.
- **Studio API** - Fix deadlock in studio when removing certain USB headsets.
- **LowLevel API** - Fixed rare case of DSP mixer not responding momentarily if output mode was switched during playback.
- **LowLevel API** - Fix declaration of `DSP.getMeteringInfo` in C# wrapper.

Notes:

- **UE4** - Now built against Unreal 4.8
09/06/15 1.06.04 - Studio API patch release

Features:

- **Studio API** - Added tempo and marker event callbacks. See FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_MARKER and FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_BEAT.
- **LowLevel API** - Add FMOD_DSP_DESCRIPTION::sys_register, sys_deregister and sys_mix for DSP plugin developers, to allow a per type (not instance) level init/shutdown/mix callback for a plugin.

Fixes:

- **Lowlevel API** - Fixed MIDI playback not correctly handling CC1 (mod wheel).
- **Studio API** - Fixed livelock when scheduling nested events in a playlist that have 0 duration.
- **Studio API** - Fixed events staying active when they have active but idle nested events on parameters. Events now finish once their nested parameter events go idle.
- **Studio API** - Fixed Scatterer sounds not being spatialized properly. Introduced in 1.06.00.
- **FSBank** - Fix bug that prevented selection of FMOD ADPCM in the tool.
- **LowLevel API** - Fix set3DPanLevel not respecting reverb mix when pan level was approaching 0.
- **LowLevel API** - Fix 2d channels not having the wet mix scaled for reverb when parent ChannelGroup volume/mute functions were called.
- **Unity** - Remove warnings about banks already loaded
- **Unity** - Fix Unity Editor crashing when using FMOD_LIVEUPDATE and FMOD_DEBUG pre-processor commands simultaneously.

Notes:

- **Studio API** - Incremented bank version, requires runtime 1.06.00 or later.
- **Studio API** - Latest runtime supports loading old bank versions from 1.03.00.
- **Studio API** - Added a warning for events that haven't finished because they
are waiting for a DSP effect to go idle.

- **Lowlevel API - WiiU** - To comply with LotCheck requirements only the logging version of FMOD will link with networking APIs
- **Xbox One** - Now built with May 2015 QFE 1 XDK.
22/05/15 1.06.03 - Studio API patch release

Features:

- *Studio API* - Reduced memory overhead of bank metadata.

Fixes:

- *Studio API* - Fixed steal oldest polyphony behaviour using instance creation instead of instance start time.
- *Studio API* - Fixed playlist instrument cutoff not applying a volume ramp down which could cause pops when using polyphony.
- *LowLevel API* - Fixed invalid memory access when loading an incorrectly formatted or truncated IMA ADPCM wav file using `FMOD_OPENMEMORY_POINT` and `FMOD_CREATECOMPRESSESAMPLE`.
- *LowLevel API* - Fix rare crash due to race condition when calling `Sound::Release()` just after the sound has been loaded using `NON_BLOCKING`.
- *LowLevel API* - *PS3* - Fix deadlock in streaming when using SPU Threads.
- *LowLevel API* - Attempting to attach the Master ChannelGroup to a port will now return `FMOD_ERR_INVALID_PARAM`.
- *LowLevel API* - *PS4* - Fix issues with recording at rates other than the driver default.
- *LowLevel API* - Fix crash in DSPFader when voice goes virtual with DSP effect added at a position lower than the fader (ie post fader), and the effect is freed.
- *UE4 Integration* - Fixed incorrect listener orientation.

Notes:

- *Studio API* - Changed envelope follower DSP sidechain parameter to ignore invalid values.
- *Xbox One* - Now built with April 2015 XDK.
- 7.1 to 5.1 downmix now lowers the volumes of the back speakers' contribution to the surround speakers by 1.5db to lessen volume bulge in the middle.
06/05/15 1.06.02 - Studio API patch release

Features:

- **Studio API** - Sounds on parameters that are cross-fading in will now apply a ramp volume up to the initial cross-fade value. This avoids pops even if the sound has been improperly authored.
- **UE4 Integration** - Added support for Reverb zones.
- **UE4 Integration** - Exposed `Studio::EventInstance::getTimelinePosition` and `Studio::EventInstance::setTimelinePosition` as blueprint functions.

Fixes:

- **Studio API** - Fixed transition timeline modules not crossfading their volume properly when overlapping the lead out region.
- **Studio API** - Fixed C# wrapper not linking against 64bit dll when WIN64 is defined.
- **Studio API** - Fixed events with sidechains never naturally stopping.
- **Studio API** - Fixed runtime crash caused by a compiler bug for customers compiling from source using VS2015 CTP.
- **LowLevel API** - **WiiU** - Fixed incorrect pitch for DRC if System is running a rate other than 48KHz or 32KHz.
- **LowLevel API** - **PS4** - Fix small memory leak when closing output ports.
- **LowLevel API** - Fixed documentation regarding `FMOD_DSP_COMPRESSOR_USESIDECHAIN` and `FMOD_DSP_ENVELOPEFOLLOWER_USESIDECHAIN`.
- **LowLevel API** - **WinStore** - Fix System::Init() freezing when called from the UI thread.
- **UE4 Integration** - Fixed bad editor performance when selecting audio components.
- **UE4 Integration** - Fixed several Android deployment issues.
- **Unity Integration** - Fix compilation issues on iOS when using IL2CPP.
- **Unity Integration** - Logging libs for OSX are now included. Defining
FMOD_DEBUG will now route FMOD internal logging to the Unity console in the OSX editor and standalone OSX builds.

- *Unity Integration* - Make members of the REVERB_PROPERTIES structure public.
- *Unity Integration* - Fix compile error on XBox One when defining FMOD_DEBUG.

**Notes:**

- *UE4 Integration* - Merged documentation with main API documentation.
17/04/15 1.06.01 - Studio API patch release

Features:

- *Studio API* - Added `FMOD_STUDIO_EVENT_CALLBACK_CREATED`, `FMOD_STUDIO_EVENT_CALLBACK_DESTROYED` and `FMOD_STUDIO_EVENT_CALLBACK_START_FAILED` callback types.
- *Studio API* - Exposed `FMOD_STUDIO_EVENT_PROPERTY_SCHEDULE_DELAY` and `FMOD_STUDIO_EVENT_PROPERTY_SCHEDULE_LOOKAHEAD` as advanced properties that can be set per instance.

Fixes:

- *LowLevel API* - Fixed Channel userdata not being reset to 0 each playSound.
- *LowLevel API* - Fixed the C# wrapper for DSP.AddInput and `DSP.disconnectFrom`.
- *LowLevel API* - PS4 - Fixed playback issue with AT9 compressed samples that have a non-zero loop start point.
- *LowLevel API* - Fix rare issues with generator DSPs or compressed channels getting stuck looping the same fragment of audio.
- *LowLevel API* - PS3 - Fix rare crash if sound stops on a channelgroup with a matrix set in it.
- *Unity Integration* - Fix banks not being loaded on standalone OSX builds.
- *Unity Integration* - Logging libs for windows are now included. Defining `FMOD_DEBUG` will now route FMOD internal logging to the Unity console in the Windows editor and standalone Windows builds.

Notes:

- *PS4* - Now built with SDK 2.508.051.
10/04/15 1.06.00 - Studio API minor release

Important:

- *Studio API* - Scatterer sounds now steal the oldest spawned sound if the polyphony limit has been reached when spawning a new sound.

Features:

- Added new compression format FADPCM as a higher quality, lower CPU cost drop in replacement for standard IMA ADPCM. This custom developed format is our new recommendation for mobile, PS Vita and Wii U delivered exclusively via FSB.
- *LowLevel API* - Improved CPU performance of convolution reverb by 30%
- *LowLevel API* - *Android* - Improved latency by automatically resampling to the native rate to enable the fast mixer.
- *LowLevel API* - *PS Vita* - Removed requirement to use 48KHz mixing, default is now 24KHz for better performance.
- *LowLevel API* - *Xbox One* - Optimized performance for looping compressed XMA.
- *LowLevel API* - *Xbox One* - Added support for greater than stereo XMA streams.
- *LowLevel API* - *Xbox One* - Reduced output latency by 20ms.
- *Studio API* - Significantly reduced memory usage.
- *Studio API* - Added Studio bank loading thread.
- *Studio API* - Added *Studio::Bank::getUserData* and *Studio::Bank::setUserData*.
- *Studio API* - Added
  - `FMOD_STUDIO_SYSTEM_CALLBACK_BANK_UNLOAD` callback.
- *Studio API* - Support for transition timeline lead-in and lead-out.
- *Studio API* - Added support for setting Studio async update period via
  - `FMOD_STUDIO_ADVANCEDSETTINGS`
- *LowLevel API* - *PS4* - Added support for High Quality Recording API by default.

Fixes:
- **Studio API** - Fixed parameter modulation being unable to go below the value set from the public API.
- **Studio API** - Fixed effect bypass setting not being saved to banks.

**Notes:**

- **Studio API** - The C# wrapper now uses System.Guid instead of FMOD.GUID.
- **Studio API** - Default command buffer size has been increased to 32K.
- **Studio API** - Changed `FMOD_STUDIO_EVENT_CALLBACK_TYPE` from an enum to a bitfield. Added 'callbackmask' parameter to `Studio::EventDescription::setCallback` and `Studio::EventInstance::setCallback`.
- **Studio API** - Deprecated Studio::ID typedef. Please use `FMOD_GUID` type.
- **Studio API** - Changed `Studio::System::setListenerAttributes` and `Studio::System::getListenerAttributes` now take a listener index in preparation for multiple listener support.
- **Studio API** - Studio::System::startRecordCommands and Studio::System::stopRecordCommands has been renamed to `Studio::System::startCommandCapture` and `Studio::System::stopCommandCapture`.
- **Studio API** - Studio::System::playbackCommands has been renamed to `Studio::System::loadCommandReplay` and now returns a CommandReplay object.
- **Studio API** - Schedule delay and look-ahead has been reduced for Studio async mode. This reduces latency and improves responsiveness of events to parameter changes.
- **LowLevel API** - `System::set3DListenerAttributes` checks for invalid arguments in release.
- **LowLevel API** - Thread safety no longer uses a command queue for deferring selected API functions until the next update. The queue of deferred functions was most often flushed by a subsequent getter function or other setter function anyway.
- **LowLevel API** - Added 'numconnected' to `System::getRecordNumDrivers` and 'state' to `System::getRecordDriverInfo` in preparation for changes to how recording device removal is handled. Currently 'numconnected' will equal 'numdrivers' and 'state' will be `FMOD_DRIVER_STATE_CONNECTED`.
- **LowLevel API** - Some of the FMOD_PAN_ types have been renamed...
FMOD_DSP_PAN_ for consistency.

- **LowLevel API** - Calling `ChannelControl::getAudibility` on a ChannelGroup now returns the combined audibility of itself and the parent audibility.
02/04/15 1.05.15 - Studio API patch release

Fixes:

- **Lowlevel API - Win** - Fixed occasional deadlock when removing audio devices.
- **Lowlevel API** - Fixed stream crackling if initialseekposition was used immediately followed by `Channel::setPosition` with a close position value.
- **Lowlevel API - PS3** - Fixed six or eight channels vorbis streams crashing the SPU.
- **Lowlevel API - Xbox One** - Fixed rare hang when decoding small XMA compressed audio.
- **Lowlevel API** - Fixed `DSP::getInfo` not returning configwidth and configheight information for VST plugins.
- **LowLevel API** - Fixed rare crash calling `DSP::reset` on the fader DSP, after `ChannelGroup::setPan/setMixLevelsOutput/setMixMatrix`. This could occur when a Studio event is started or stopped.
- **Lowlevel API - PS4** - Fixed issue with non-1024 sample aligned loop-points and AT9 compressed samples.
- **Lowlevel API** - Fixed DSP effect being un-removable with `FMOD_ERR_DSP_INUSE` after being added to a channel and the channel stops (becoming invalid).

Notes:

- **PS3** - Now built with SDK 470.001.
 Fixes:

- **Lowlevel API** - Fixed some cases where channel group audibility was not refreshed when fade points are active. This could happen when a Studio event instance is paused and unpaused.
- **Lowlevel API - PS3** - Fixed FMOD_PS3_INITFLAGS overlapping **FMOD_INITFLAGS** causing certain **FMOD_INITFLAGS** to affect PS3 specific bit-stream encoding options.
- **Lowlevel API - PS3** - Fixed a rare hang when releasing a DSP that exposes a **FMOD_DSP_PARAMETER_OVERALLGAIN** parameter.
- **Lowlevel API - PS3** - Fixed opening URL failing with network streams.
- **Lowlevel API - Xbox One** - Fixed recording API, you can now specify any sample rate to record at. Native rate of 48KHz is still recommended for lowest latency.
- **Studio API** - Fixed virtualized event failing to become real if the number of playing instances drops back below max polyphony.

 Notes:

- **Lowlevel API - PS3** - Deprecated **FMOD_PS3_EXTRADRIVERDATA::initflags**. Due to a bug it was being ignored. Pass the flags to **System::init** instead.
25/02/15 1.05.13 - Studio API patch release

Features:

- Studio API - Studio::System::getBank now accepts bank filenames as input.
- LowLevel API - Added 64bit versions of fsbank and fsbankcl tools.

Fixes:

- Studio API - Fixed case of nested event not being destroyed even after it had finished producing sound.
- Studio API - Fixed crash when changing output bus on an event when live update is connected.
- Studio API - Fixed deadlock when calling Studio commands when mixer is suspended.
- Studio API - System::release now ensures release even if flushing pending commands causes an error.
- Studio API - The name of the string bank is now added to the string bank.
- Studio API - Event instance restart now flushes parameter values before timeline rescheduling occurs. This avoids a potential issue for transitions with parameter conditions, where they may not have used the most recent parameter value.
- LowLevel API - Fix unnecessary querying of recording driver capabilities during recording.
- LowLevel API - PS4 - Remove AT9 workaround added in 1.05.12. Fix AT9 compressed codecs with finite loop counts.
- LowLevel API - PS4 - Removed stalls when an AT9 compressed sample channel is started.
- LowLevel API - PS4 - Fix crash in recording.
- LowLevel API - Fix multiple listener support not working properly.
- LowLevel API - Fix user file crash if using asyncfileread callback, and file open and close happens without any read occurring.
- LowLevel API - Fix rare crash with Sound::release if a nonblocking setPosition is in process and it is an FSB sound.
- LowLevel API - Fix thread safety issue loading multiple mod/s3m/xm/it files simultaneously.
- LowLevel API - Fix rare crash if FMOD_ACCURATETIME was used with
.mp3 file followed by mp3 encoded FSB file after a time, both using \texttt{FMOD\_CREATECOMPRESSESAMPLE}.

- \textit{LowLevel API - PS3} - Fixed custom DSP that use the plugindata member.

\textbf{Notes:}

- \textit{Studio API} - Updated programmer\_sound example.
- \textit{LowLevel API} - Added plug-in inspector example.
- \textit{Xbox One} - Now built with February 2015 XDK.
06/02/15 1.05.12 - Studio API patch release

Features:

- Studio API - Improved memory use for events with repeated nested events.

Fixes:

- Studio API - Fixed crash when stopping multiple one-shot events on a bus.
- Studio API - Fixed nested event polyphony from cutting off immediately, causing pops.
- LowLevel API - Fixed rare crash in mixer if DSP::reset is called on a FMOD_DSP_TYPE_FADER dsp (ie ChannelControl head DSP) after DSP::setPan/setMixLevelsOutput/setMixMatrix.
- LowLevel API - Fixed race condition setting resampling speed while resampling is occurring.
- LowLevel API - Fixed crash loading mod/s3m/xm/it file using FMOD_NONBLOCKING and FMOD.CreateCompressedSample and then immediately calling SoundI::release
- LowLevel API - PS4 - Work around AT9 codec issues that have appeared when using SDK 2.000
22/01/15 1.05.11 - Studio API patch release

Features:

- Lowlevel API - Xbox One - Added access to 7th CPU core.

Fixes:

- Studio API - Fixed crash setting Studio event callback to null as it is being invoked.
- Studio API - Fixed crash in hashmap reallocation when running out of memory.
- Studio API - Fixed Studio::loadBankCustom from freeing its custom userdata before the close callback for failed banks.
- Studio API - If FMOD_OUTTPUTTYPE_WAVWRITER_NRT or NOSOUND_NRT is used as an output mode, Studio runtime will now internally force FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE to avoid a hang.
- LowLevel API - Fix pop noise when 3d sound goes virtual then becomes real again, only if ChannelControl::addDSP was used.

Notes:

- Studio API - Studio::EventInstance::getPlaybackState will now return the state as FMOD_STUDIO_PLAYBACK_STOPPED if the instance is invalid.
- Studio API - Studio::Bank::getLoadingState, Studio::Bank::getSampleLoadingState, and Studio::EventDescription::getSampleLoadingState will now return the state as FMOD_STUDIO_LOADING_STATE_UNLOADED if the object is invalid.
Features:

- *LowLevel API - Xbox One* - Added support for recording from microphones.

Fixes:

- *Lowlevel API - PS3* - Fix deadlock in streaming sounds when linking against the SPU thread libraries.
- *Lowlevel API - Windows* - Fixed crash when initializing a second ASIO system.
- *Lowlevel API - Fix ChannelControl::getDSPIndex* not returning an error if the dsp did not belong in the Channel or ChannelGroup
- *Lowlevel API - PS3* - Fix linker error when using libfmod_sputhreads.a
- *Lowlevel API - Fixed AT9 and ACP XMA incorrectly allowing FMOD_OPENMEMORY_POINT and FMOD_CREATESAMPLE.*
- *LowLevel API - Fixed reverb wetlevel from ChannelControl::setReverbProperties* being reset after a voice goes virtual then returns as real.
- *Lowlevel API - Fixed removing/adding DSPs in a ChannelControl chain copying setDelay commands incorrectly and making channels pause when they shouldnt
- *Studio API - Reinstated support for embedded loop points in source sounds, and fixed issue with playback of timelocked sounds.*
12/12/14 1.05.09 - Studio API patch release

Features:

- LowLevel API - Added System::registerOutput to allow the creation of statically linked custom output modes.

Fixes:

- Studio API - Fixed looping event cursor position from getting slightly out of sync with scheduled position when running at 44.1kHz.
- Studio API - Fixed playlist steal-oldest polyphony returning errors when used with extremely small durations.
- Studio API - Fixed silence after unpausing an event instance. Introduced in 1.05.08.
- Lowlevel API - Fixed crash after setting the input format of a Return DSP to stereo, when the mixer output format is mono.
- Lowlevel API - Windows - Fix crash calling FMODASYNCREADINFO::done function pointer when default calling convention is not cdecl.
- Lowlevel API - Fixed FMOD_SPEAKERMODE_RAW panning incorrectly during an up or down mix.
- Lowlevel API - Fix crash when a Channel::setPosition is called on a non-blocking stream that is going virtual.
- Lowlevel API - Fixed potential audio corruption when playing sounds with more than 8 channels.
- Lowlevel API - Fixed emulated channels not updating their parent when changing channel group parents.
- Lowlevel API - Fixed channels having their volume reset when changing channel group parents.
- Lowlevel API - Fixed channels not going virtual when being paused if no other volume changes were occurring.
- Lowlevel API - Fixed FMOD_INIT_PROFILE_ENABLE enabling all DSP metering regardless of whether FMOD_INIT_PROFILE_METER_ALL was used.
- Lowlevel API - Added volume ramp up for 3d channels coming back from virtual to real.
• *Lowlevel API* - Fixed rare crash if the master channelgroup's DSP Head unit was changed then released.
• *Lowlevel API - PS3* - Fix loudness meter not functioning.
• *LowLevel API - PS3* - Re-enabled playDSP.

**Notes:**

• *Lowlevel API - Xbox One* - Added check to ensure DSP buffer size is specified as the default 512 samples to avoid performance degradation of XMA decoding.
• *Lowlevel API - Windows* - Changed default ASIO speaker mapping to 1:1 for outputs with more than 8 channels.
28/11/14 1.05.08 - Studio API patch release

Features:

- **LowLevel API** - Added `ChannelControl::setFadePointRamp` helper function that automatically adds a volume ramp using fade points.

Fixes:

- **Studio API** - Fixed pops that could occur when stopping events that have instruments scheduled to stop already.
- **Studio API** - Fixed pop that could occur when playing an sound that has an AHDSR fade in.
- **Studio API** - Fixed indeterminism when exporting string banks that have multiple string entries with inconsistent case.
- **Lowlevel API - Android** - Improved compatibility with Java 1.6.
- **Lowlevel API - iOS** - Added armv7s back to the universal binary.
- **Lowlevel API - Windows** - Fix `System::getRecordDriverInfo` returning incorrect device names. Introduced in 1.05.00.
- **Lowlevel API** - Fix `FMOD_SPEAKERMODE_RAW` creating silence if playing more than 1 sound, introduced in 1.04.15.
- **LowLevel API** - Fix CELT and Vorbis FSB being allowed with `FMOD_OPENMEMORY_POINT` and `FMOD_CREATESAMPLE` when it should in fact return `FMOD_ERR_MEMORY_CANTPOINT`
21/11/14 1.05.07 - Studio API patch release

Important:

- *Studio API* - Fixed scheduling for looping nested events which are cut off from the parent event. Previously the looping nested event would attempt to play to end but would cut off halfway through with a noticeable click. The nested event now cuts off immediately with a fade out ramp.

Features:

- *LowLevel API - Android* - Added ARM64 support.

Fixes:

- *LowLevel API* - Fix `ChannelGroup::setMixLevelsOutput` causing corrupted audio when the channel group has an input with a channel count greater than the system channel count.
- *Lowlevel API* - Fixed `ChannelControl::setMute` not refreshing channel audibility.
- *Lowlevel API - Windows* - Fix calls to `DSP::getInfo()` from within a custom DSP deadlocking the system during audio device change.
- *Lowlevel API* - Fixed some DSP configurations with an idle first input causing signal to be downmixed to mono.
- *Lowlevel API* - Fix invalid characters being printed in log messages when open certain types of media files.
- *Studio API* - Fixed `Studio::Bank::getEventCount` and `Studio::Bank::getEventList` incorrectly enumerating nested events.
- *Studio API* - Fixed `Studio::Bus::stopAllEvents` not working when snapshots or nested events are playing.
- *Studio API* - Fixed "state->mInstanceCount > 0" assert that could occur when instruments were stopped repeated times.

Notes:

- *PS Vita* - Now built with SDK 3.300.
- *WiiU* - Now built with SDK 2.11.13.
13/11/14 1.05.06 - Studio API patch release

Features:

- LowLevel API - Windows - Add 64bit VST support

Fixes:

- Studio API - Fixed thread safety issue when accessing sound tables as new banks with sound tables are being added.
- Studio API - Fixed duplicate streaming sounds keeping playing when unloading the memory bank that it is streaming from. Now the stream will stop if the memory bank is unloaded.
- Lowlevel API - Fixed sound going mono in certain DSP configurations. Introduced 1.05.00.
- Lowlevel API - Fixed rare timing issue with fade points that caused the fader DSP to generate invalid floats.
- Lowlevel API - Android - Fixed crashes due to insufficient stack size when running ART instead of Dalvik.
- Lowlevel API - Windows - Fixed potential crash if using multiple FMOD::System with ASIO output mode. This is not supported by ASIO and is now disabled.
- Lowlevel API - Linux - Fixed PulseAudio device enumeration not placing the default device at position 0.

Notes:

- Xbox One - Now built with November 2014 XDK.
- Android - Now built with NDK r10c.
- iOS - Now built with SDK 8.1 (Xcode 6.1).
- Mac - Now built with SDK 10.10 (Xcode 6.1).
- PS4 - Now built with SDK 2.000.071
30/10/14 1.05.05 - Studio API patch release

Features:

- *LowLevel API* - Added convolution reverb example.

Fixes:

- *Studio API* - Fixed various errors after deleting objects via Live Update
- *Studio API* - Fixed possible crash using shared waveforms across multiple banks after some of the duplicate banks have been unloaded.
- *Studio API* - Fixed duplicate events becoming invalidated when the first bank that contains them is unloaded.
- *Studio API* - Fixed bug in load_banks example.
- *Studio API* - Fixed crash when calling `Studio::Bus::stopAllEvents` while playing event instances that have had release called.
- *LowLevel API* - Fixed memory leaks.
22/10/14 1.05.04 - Studio API patch release

Fixes:

- *Studio API* - Fixed transition markers failing at the start of the timeline.
- *LowLevel API* - Fixed incorrect behaviour when changing the channel mode from 3D to 2D.
- *FSBank* - Fixed setting the cache directory and printing log messages.

Notes:

- *Studio API* - Timeline transitions can be chained together with no delay.
13/10/14 1.05.03 - Studio API patch release

Features:

- Studio API - Added Studio::Bus::lockChannelGroup and Studio::Bus::unlockChannelGroup.

Fixes:

- Studio API - Fixed event fadeout cutting off DSP effects with long tails.
- Studio API - Fixed crash when accessing events with a lifetime across duplicate banks. Note that when unloading the initial bank for an event, that event will be invalidated.
- Lowlevel API - Fixed for enum mismatches in C# wrapper.
- Lowlevel API - Fixed FMOD_DSP_LOWPASS click when being reused, ie stopping then starting a new sound.
- Lowlevel API - Fixed rare hang during Sound::release for sounds created as FMOD NONBLOCKING.
- Lowlevel API - Fixed corrupted playback of MOD/S3M/XM/IT/MID sequenced formats. Introduced 1.04.05
- Lowlevel API - Fixed ChannelGroup::setReverbProperties on the master channel group causing a stack overflow. This is now disallowed as it creates a circular dependency.
- Lowlevel API - Mac - Replaced error condition if initializing FMOD before activating the AudioSession with a TTY warning.
- Lowlevel API - Android - Fixed Sound::readData going forever when reading AAC audio.
- Lowlevel API - Fix the pancallbacks member of the FMOD_DSP_STATE_SYSTEMCALLBACKS structure passed into custom DSP callbacks being NULL.
- Lowlevel API - Fix crash in mixer if user used System::playDSP and made the dsp inactive with DSP::setByPass or DSP::setActive
01/10/14 1.05.02 - Studio API patch release

Fixes:

- **Studio API** - Fixed a bug where event sounds could have incorrectly synchronized playback.
- **Lowlevel API** - Fixed compiler warning in public header.
- **Studio API** - Disabled embedded loop points on all sounds played by the Studio API to fix incorrect playback of timelocked sounds.
- **Studio API** - Snapshot volumes are now interpolated in terms of linear gain.
- **LowLevel API** - Fixed 3EQ DSP not clearing out internal state when `DSP::reset` is called, potentially causing audible artifacts when reused.
- **LowLevel API** - Fixed resampler inaccuracies when reading partial looping data.
- **LowLevel API - Mac** - CoreAudio output mode will now allow any DSP block size and will correctly use the desired buffer count.
- **LowLevel API - Mac** - CoreAudio now correctly exposes its output AudioUnit via the `System::getOutputHandle` function.
- **LowLevel API - PS3** - Fixed resampler strict aliasing bug on PPU release builds.
- **LowLevel API - PS3** - Fixed playDSP crash.
- **LowLevel API** - Fixed convolution reverb input downmix
- **Lowlevel API** - Greatly increase speed of mixing multichannel source data.

Notes:

- **Studio API** - Loudness meters saved in banks will now be bypassed to improve performance when profiler isn't attached.
- **Xbox One** - Now built with September 2014 QFE 1 XDK.
22/09/14 1.05.01 - Studio API patch release

Important:

- Fixed possible large memory blowout if certain DSP pools were exceeded.

Features:

- LowLevel API - Added FMOD_SYSTEM_CALLBACK_PREUPDATE and FMOD_SYSTEM_CALLBACK_POSTUPDATE callbacks.
- LowLevel API - iOS - Added support for multichannel output via accessory.

Fixes:

- Studio API - Implemented Studio::Bus::stopAllEvents.
- Studio API - Fixed bus going silent when output format is stereo and system speaker mode is 5.1.
- Studio API - Fixed Studio::System::getAdvancedSettings clearing FMOD_STUDIO_ADVANCEDSETTINGS.cbSize and not returning default values.
- Studio API - Fixed a bug where looping or sustaining events could incorrectly be treated as oneshot.
- Studio API - Fixed a bug where event sounds could have incorrectly synchronized playback.
- LowLevel API - Fix bug with custom 3D rolloff curve points getting corrupted.
- LowLevel API - Fixed incorrect documentation for FMOD_CHANNELCONTROL_CALLBACK.
- LowLevel API - Fix sub-mix channel count being incorrect when playing multiple sounds with different channel counts together. Introduced in 1.04
- LowLevel API - Fix a channel DSP head's ChannelFormat not being reset if a sound with a channel mask was played on it previously

Notes:

- iOS - Now built with SDK 8.0 (Xcode 6.0).
- Mac - Now built with SDK 10.9 (Xcode 6.0).
09/09/14 1.05.00 - Studio API minor release

Important:

- **Studio API** - Studio::MixerStrip has been replaced by Studio::Bus and Studio::VCA. The new classes do not have a release method, as their handles remain valid until the bank containing them is unloaded.
- **Studio API** - Studio::System::getEvent, Studio::System::getBus, Studio::System::getVCA, and Studio::System::getBank now take a string path. Functions that take an ID are now named Studio::System::getEventByID, Studio::System::getBusByID, Studio::System::getVCAById, and Studio::System::getBankByID.
- **Studio API** - Studio::EventInstance::release no longer invalidates the handle immediately. The handle remains valid until the event stops and is actually destroyed. In addition, this function now succeeds even if the event will not stop naturally.
- **Studio API** - Events with sounds triggered by parameters will now stop rather than going idle when all sounds have finished and the timeline cursor has reached the end.
- **LowLevel API** - The wide string argument of System::getDriverInfo() and System::getRecordDriverInfo() has been removed. These functions now return UTF-8 encoded strings.
- **LowLevel API** - FMOD_UNICODE flag for System::createSound() and System::createStream() has been removed. These functions now accept UTF-8 encoded file names to load sounds with non-ASCII characters on Windows, PS3, PS4, PS Vita, XBox One, iOS, Android, Mac, Linux, Windows Phone, and Windows Store.
- **FSBank API** - FSBANK_INIT_UNICODE flag has been removed. All file name structure members and function arguments now accept UTF-8 encoded strings.

Features:

- **Studio API** - Added FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_CREATED and FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_DESTROYED callback types.
- **Studio API** - Added `Studio::Bank::getStringCount` and `Studio::Bank::getStringInfo`.
- **LowLevel API** - Added `FMOD::Debug_Initialize` to configure the destination and level of debug logging when using the logging version of FMOD. This is a more flexible replacement for the previous `FMOD::Debug_SetLevel API`.
- **LowLevel API** - **Android** - Added support for playing AAC files (requires Android 4.2).
- **LowLevel API** - **Android** - Reduced latency for devices that support FastMixer, see "Basic Information" section of the docs CHM for details.
- **LowLevel API** - Added `DSP::setWetDryMix` so any effect can have generic wet/dry signal level control.
- **LowLevel API** - Added 3D ChannelGroup support. A ChannelGroup is a 'group bus' and it can now be positioned in 3d space, affecting the mix for buses and channels below it. 3D Cones, geometry etc all work. Use `ChannelGroup::setMode(FMOD_3D)` to enable a 3D ChannelGroup.

**Fixes:**

- **Studio API** - Fixed transitions to the end of a loop region failing to escape the loop.
- **Studio API** - Fixed sends inside events occasionally having a brief period of full volume when the event is started.
- **LowLevel API** - Fix for crash when creating multiple Systems with profiling enabled.

**Notes:**

- **Studio API** - Transition regions no longer include the end of their range.
- **Studio API** - `FMOD_ERR_EVENT_WONT_STOP` has been removed.
- **Studio API** - `Studio::EventInstance::start` now resets all DSPs inside the event to prevent incorrect ramping. This relies on `FMOD_DSP_RESET_CALLBACK` leaving public parameters unchanged.
- **Studio API** - `Studio::System::getEvent`'s unimplemented mode parameter has been removed, along with `FMOD_STUDIO_LOADING_MODE`.
- **Studio API** - `FMOD_STUDIO_PLAYBACK_IDLE` and `FMOD_STUDIO_EVENT_CALLBACK_IDLE` have been removed.
- **Studio API** - Removed deprecated function
Studio::EventInstance::createSubEvent.

- **LowLevel API** - Changed `FMOD_FILEASYNCCANCEL_CALLBACK` to take `FMODASYNCREADINFO` structure rather than `void *handle`.
- **LowLevel API** - The `FMOD_DSP_RESET_CALLBACK` documentation has been updated to make it clear that it should leave public parameters unchanged.
- **LowLevel API** - **PS4** - Thread affinity can now be set as a mask of allowed cores.
- **LowLevel API** - **iOS / Mac** - Reduced default block size to 512 for lower latency.
- **LowLevel API** - **Android** - Renamed Java interface from `FMODAudioDevice` to simply `FMOD`, i.e. `org.fmod.FMOD.init()`.
- **LowLevel API** - **Channel::setMode** moved to `ChannelControl::setMode` so that ChannelGroups can now also have 2D/3D mode bits set.
09/09/14 1.04.08 - Studio API patch release

Fixes:

- **Studio API** - Fixed crash when a second system is created but never initialized.
- **Studio API** - Fixed a few ordering issues that can cause binary changes in identical banks.
- **LowLevel API** - Fix for pop due to unwanted volume ramping after calling `DSP::reset` on fader DSP. This can happen when a Studio event instance is stopped and then restarted. Caused by timing issue with `ChannelControl::setVolumeRamp`.
- **LowLevel API** - Fix crash if `FMOD_DSP_TYPE_MIXER` or DSP with no read/process function is passed to `System::playDSP`.
- **LowLevel API - iOS** - Fixed MP2 files being intercepted by AudioQueue causing an internal error.
- **LowLevel API - XboxOne** - Fixed network connect not resolving host names and not honoring the requested time out.
- **LowLevel API** - Fix rare crash if calling `Channel::stop()` which a non blocking `Channel::setPosition` is happening with a stream.
- **LowLevel API - Winphone and Windows Store Apps** - fixed detection of socket errors.
- **LowLevel API** - If a user DSP is added after the master ChannelGroup's fader that changes the output channel count to something other than the software mixer's channel count, stuttering could occur. Fixed.
- **Studio API - Windows** - Fixed Studio API functions deadlocking when in asynchronous mode and a sound device is removed.
- **LowLevel API - PS3** - Fixed some DSP parameter sets being ignored (overwritten by SPU DMA)
- **LowLevel API** - Fix `System::playDSP` not working with custom DSP effects that use the read callback
- **LowLevel API** - Added `Memory.Initialize` function to the C# wrapper.
- **LowLevel API** - Fixed incorrect truncation of `FMOD_CREATECOMPRESSESAMPLE` sounds created from MP3 or MP2 files (does not affect FSB).
- **LowLevel API** - Fixed `FMOD_ACCURATETIME` for `FMOD_CREATECOMPRESSESAMPLE` sounds created from MP3 or
MP2 files (does not affect FSB).

**Notes:**

- *LowLevel API* - The master ChannelGroup's fader/head now is not responsible for up or downmixing the signal. It is now done beyond the dsp tree, internally within FMOD. The Master ChannelGroup's Fader can still be forced to upmix if required with `DSP::setChannelFormat`. 
20/08/14 1.04.07 - Studio API patch release

Fixes:

- *Studio API* - Fixed an internal error when instantiating a snapshot that exposes intensity as a parameter.
- *LowLevel API - PS3* - Fix audio dropout/corruption when using `FMOD_SPEAKERMODE_STEREO`. Usually when sidechain is involved.
- *LowLevel API - iOS* - Fixed `System::recordStart` returning `FMOD_ERR_RECORD`.
- *LowLevel API* - Fixed possible click noise on end of sound if using `FMOD_CREATECOMPRESSESAMPLE` or PCM on PS3.

Notes:

- *Xbox One* - Now built with July 2014 QFE1 XDK.
- *PS4* - Now built with SDK 1.750.061
06/08/14 1.04.06 - Studio API patch release

Features:

- **Studio API** - Added `FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES.subsoundIndex` to support non-blocking loading of FSB subsounds.
- **LowLevel API - Windows** - FMOD now handles sound card removal and insertion without any programmer intervention. If `System::setCallback` is called with `FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED` bit set, this feature is disabled.
- **LowLevel API - Windows** - `System::setOutput` can be called post-init now, allowing dynamic switching between any output mode at runtime.
- **LowLevel API - iOS** - Improved IMA ADPCM decoding performance especially for arm64 variant.

Fixes:

- **LowLevel API** - Fixed bug where channels with post-fader DSP units would not play after transitioning from virtual to real.
- **LowLevel API** - Fix pops with resampler on loops.
- **LowLevel API** - Fix playDSP returning `FMOD_ERR_DSP_SILENCE` or `FMOD_ERR_DSP_DONTPROCESS` if the dsp played returned that during query mode.
- **LowLevel API - PS3** - Fix ITEcho, SFXReverb, FFT DSP effects rarely getting parameters reset to old values.
- **LowLevel API - Xbox One** - Fixed audio corruption when playing mono streams from an XMA FSB that has both mono and stereo subsounds.
- **LowLevel API - PS3** - Moved vorbis decode work during stream setPosition to the SPU.
- **LowLevel API - Windows** - Fix `System::setSoftwareFormat` with differing samplerate and speaker mode causing static.
- **FSBank API** - Fixed cache being incorrectly reused when replacing source files of identical name with a different file that has an old time stamp.

Notes:
• PS3 - Now built with SDK 460.001
25/07/14 1.04.05 - Studio API patch release

Fixes:

- **Studio API** - Fix for pop when stopping events with `FMOD_STUDIO_STOP_ALLOWFADEOUT`.
- **Studio API** - Fix incorrect scheduling when Multi Sounds contain nested events with timeline transitions.
- **Studio API** - Fix for nested event modules not being cleaned up when used in a playlist module.
- **Studio API** - Fix for events failing to stop when they have instruments on parameters that have Hold enabled.
- **LowLevel API** - Fix combined volume ramp and fade point ramp having an incorrect volume.
- **LowLevel API** - Fix for pop when setting zero volume with vol0virtual enabled.
- **LowLevel API** - Fix for pop when scheduling fade point ramps in the past.
- **LowLevel API** - Fix for pop on a return bus when all incoming sends go idle.
- **LowLevel API** - Fix for faders delaying going idle for a mix when volume is set to zero without ramping.
- **LowLevel API** - Fixed FSB forwards compatibility issue causing load failures.
- **LowLevel API - XboxOne** - Fixed ACP race condition when shutting down / initializing System causing XMA channels to not play.
- **LowLevel API - XboxOne** - Fixed XMA compressed sample channel leak that would eventually result in all sounds playing emulated (silent).
- **LowLevel API - WinPhone and WSA** - Fixed network connection not respecting system timeout value.
- **LowLevel API - PS3** - Fix IMA ADPCM support for ps3 not working.
- **LowLevel API - iOS** - Fixed potential duplicate symbol error on link.

Notes:

- **Xbox One** - Now built with July 2014 XDK.
- **iOS** - Now built with SDK 7.1 (Xcode 5.1.1).
11/07/14 1.04.04 - Studio API patch release

Fixes:

- *Studio API* - Fixed a bug where quantized timeline transitions could fail to play the audio at the destination marker.
- *LowLevel API* - Fix channel stealing not calling end callback in playDSP case.
- *LowLevel API* - Fix rare crash if System::playDSP is called, and it steals a channel with a sound on it, and System::update wasnt called in between.
- *LowLevel API - Mac, iOS and Android* - Improved network error reporting.
08/07/14 1.04.03 - Studio API patch release

Features:

- **LowLevel API** - Added `FMOD_ADVANCEDSETTINGS.randomSeed`, which specifies a seed value that FMOD will use to initialize its internal random number generators.

Fixes:

- **Studio API** - Changed `isValid()` functions in C# wrapper to call through to C++.
- **Studio API** - Fixed `Studio::System::loadBankCustom` file callbacks being ignored when `System::setFileSystem` is using async read functions.
- **Studio API** - Fixed incorrect playback when starting an event instance immediately after creating it.
- **Studio API** - Fixed some snapshot types causing silence on busses if loading files older than those created with FMOD Studio 1.04.02.
- **LowLevel API** - Fixed incorrect `FMODASYNCREADINFO.offset` value passed to `FMOD_FILEASYNCREAD_CALLBACK` when file buffering is disabled.
- **LowLevel API** - Windows - Fixed WASAPI failing to initialize on certain old drivers.
- **LowLevel API** - Fix `FMOD_SPEAKERMODE_RAW` being broken.
- **LowLevel API** - Fixed `System::set3DRolloffCallback` not working.
- **LowLevel API** - Fixed `Sound::set3DCustomRolloff` not working.
- **LowLevel API** - Fix Geometry API not running in its own thread like it was in FMOD Ex.
- **LowLevel API** - Fixed incorrect DSP clock when re-routing a ChannelGroup immediately after adding DSPs to it.
- **Profiler** - Fixed nodes appearing to overlap each other if multiple outputs were involved.
- **LowLevel API** - PS4 - Fixed unnecessary file reads when playing an AT9 stream.
27/06/14 1.04.02 - Studio API patch release

Fixes:

- **Studio API** - Fixed incorrect virtualization of events that have more than one 3D position dependent effect.
- **Studio API** - `Studio::EventDescription::getInstanceList`, `Studio::Bank::getEventList`, `Studio::Bank::getMixerStripList` and `Studio::System::getBankList` now accept a capacity of 0.
- **LowLevel API** - Fixed a race condition that could lead to DSP graph changes not being handled correctly.
- **LowLevel API - Linux** - Fixed "spurious thread death event" messages appearing when attached with GDB.
- **LowLevel API - Linux** - Fixed internal PulseAudio assert if `System::getNumDrivers` or `System::getDriverInfo` is used before `System::init`.
- **LowLevel API - Linux** - Fixed ALSA not using correct default driver in some cases.
- **LowLevel API** - Send levels set before connecting the Return DSP are now applied immediately rather than fading in over a short time.
Features:

- **Studio API** - Added event playback states `FMOD_STUDIO_PLAYBACK_STARTING` and `FMOD_STUDIO_PLAYBACK_STOPPING`. `FMOD_STUDIO_PLAYBACK_STARTING` will be returned after `Studio::EventInstance::start` until the event actually starts. `FMOD_STUDIO_PLAYBACK_STOPPING` will be returned after `Studio::EventInstance::stop` until the event actually stops.

- **LowLevel API** - PS4 - Added support for background music that cannot be broadcast.

Fixes:

- **Studio API** - Fixed playlist module upmixing to system speakermode when playing multiple overlapping sounds.

- **LowLevel API** - Fix streams opened with `FMOD_NONBLOCKING` from playing at the incorrect position if they go virtual shortly after setPosition is called.

- **LowLevel API** - Fix EOF detection in file system causing rare extra file read past end of file.

- **LowLevel API** - Fix some `FMOD_CREATECOMPRESSEDSOUND` based samples finishing early if they were playing back at a low sample rate.

- **LowLevel API** - Fix a bug where DSP nodes could get stuck in an inactive state after setting pitch to 0.
11/06/14 1.04.00 - Studio API minor release

Important:

- Added PS3 platform support.
- Added WiiU platform support using SDK 2.10.04.
- Added Linux platform support.
- Added Windows Phone 8.1 platform support.
- FMOD_HARDWARE and FMOD_SOFTWARE flags have been removed. All voices are software mixed in FMOD Studio.

Features:

- Studio API - Added Studio::System::getSoundInfo for accessing sound table entries
- Studio API - Added Studio::EventInstance::setProperty, Studio::EventInstance::getProperty and FMOD_STUDIO_EVENT_PROPERTY_CHANNELPRIORITY
- Studio API - Added doppler effect support
- Studio API - Added libfmodstudio.a import library for MinGW/Cygwin. C API only, C++ linking not supported.
- LowLevel API - Improved low level DSP mixing performance by about 30%
- LowLevel API - Added new System callback type FMOD_SYSTEM_CALLBACK_THREADDESTROYED.
- LowLevel API - System::attachChannelGroupToPort now has an argument to allow signal to be passed to main mix.

Fixes:

- Studio API - Fixed nested events never ending if they have a parameter with non-zero seek speed.
- Studio API - Fixed AHDSR modulation on snapshot intensity
- LowLevel API - Fixed incorrect fader interpolation when reparenting channels with propagate clocks.
- LowLevel API - Win - Fixed several issues with ASIO playback.
- LowLevel API - Android - Fixed audio corruption on devices without
NEON support.

- **LowLevel API** - Fixed `FMOD_CREATESOUNDEXINFO.length` being handled incorrectly for memory sounds. This length represents the amount of data to access starting at the specified fileoffset.
- **LowLevel API** - Fix truncated FSB causing zero length subsounds, now returns `FMOD_ERR_FILE_BAD`

**Notes:**

- **LowLevel API** - Renamed C# wrapper `SYSTEM_CALLBACKTYPE` to `SYSTEM_CALLBACK_TYPE` so it matches the C++ API.
- **LowLevel API** - Renamed MinGW/Cygwin import library to `libfmod.a`.
- **Studio API** - Deprecated C# wrapper functions `Studio.Factory.System_Create` and `Studio.System.init` have been removed. Use `Studio.System.create` and `Studio.System.initialize` instead.
- **Studio API** - Deprecated function `Studio::EventInstance::getLoadingState` has been removed. Use `Studio::EventDescription::getSampleLoadingState` instead.
- **Studio API** - `FMOD_STUDIO_EVENT_CALLBACK` now takes an `FMOD_STUDIO_EVENTINSTANCE*` parameter.
- **Xbox One** - Now built with June 2014 XDK.
29/05/14 1.03.09 - Studio API patch release

Fixes:

- **Studio API** - Fixed truncation error when loading sample data from bank opened with `Studio::System::loadBankMemory`.
- **Studio API** - Fixed numerical error when blending multiple snapshots with zero intensity.
- **Studio API** - Fixed incorrect pitch when an instrument has a non-zero base pitch combined with pitch automation or modulation.
- **LowLevel API - Xbox One** - Fixed potential seeking inaccuracies with XMA sounds.
- **LowLevel API** - Fix occasional audio pops when starting a channel or channel group.
- **LowLevel API** - Fix crash when running out of memory during channel group creation.
- **LowLevel API** - Fixed the C# wrapper for `Sound.setDefaults` and `Sound.getDefaults`.

Notes:

- **Studio API** - C# wrapper now takes care of setting the `FMOD_STUDIO_ADVANCEDSETTINGS.cbSize` and `FMOD_STUDIO_BANK_INFO.size` fields.
- **LowLevel API - C#** wrapper now takes care of setting the `FMOD_ADVANCEDSETTINGS.cbSize` and `FMOD_CREATESOUNDEXINFO.cbsize` fields.
- **PS Vita** - Now built with SDK 3.150.021.
Features:

- *LowLevel API* - Add `FMOD_INITASYNCREAD_FAST` and `FMODASYNCREADINFO.done` method, to improve performance of asyncread callback significantly. Instead of setting 'result', call 'done' function pointer instead.
- *LowLevel API* - Multiple channel groups can now be attached to the same output port.

Fixes:

- *LowLevel API - Windows Store* - Fixed `System::getRecordDriverInfo()` returning incorrect number of channels.
- *LowLevel API* - Fix `System::setFileSystem` asyncread callback not setting priority values properly.
- *LowLevel API* - Releasing a channel group attached to an auxiliary output port now cleans up resources correctly.
- *LowLevel API* - Channel groups attached to an auxiliary output port can now be added as children of other channel groups.
- *LowLevel API* - Fix `DSPConnection::setMix()` not being applied properly.
- *LowLevel API - PS Vita* - Fixed potential crash during `System::init` if any output related pre-init APIs are used, such as `System::getNumDrivers`.
- *LowLevel API - PS Vita* - Fixed crash if an attempt is made to load AT9 FSBs as a compressed sample, for PS Vita this is a streaming only format.
- *LowLevel API - Xbox One* - Small improvement to XMA performance.
- *LowLevel API* - Fixed the C# wrapper for `System.playDSP`.
- *LowLevel API* - Fixed potential crash on ARM platforms when loading an FSB.

Notes:

- *PS4* - Now built with SDK 1.700.
• Android - Now built with NDK r9d.
Features:

- *Studio API* - Improved performance for projects with many events.
- *Studio API* - Improved memory usage for projects with many bus instances.
- *Studio API* - Added `Studio::System::setCallback`, `Studio::System::setUserData` and `Studio::System::getUserData`.
- *LowLevel API* - Added `gapless_playback` example for scheduling/setDelay usage.
- *LowLevel API* - Improved performance of logging build.
- *LowLevel API* - Added `FMOD_SYSTEM_CALLBACK_MIDMIX` callback.

Fixes:

- *Studio API* - Fixed AHDSR Release not working when timelocked sounds are stopped by a parameter condition.
- *Studio API* - Removed some unnecessary file seeks.
- *Studio API* - Fixed AHDSR Release resetting to Sustain value when instruments with limited Max Voices are stopped repeatedly.
- *Studio API* - Fixed channels within paused events not going virtual.
- *Studio API* - Fixed AHDSR Release not working inside nested events
- *LowLevel API* - Fixed downmixing to a quad speaker setup.
- *LowLevel API* - Fixed fsb peak volume levels on big endian platforms.
- *LowLevel API* - Fixed paused channels not going virtual.
17/04/14 1.03.06 - Studio API patch release

Features:

- Studio API - Improved performance of automation and modulation

Fixes:

- Studio API - Fixed crash when creating new automation via LiveUpdate
- Studio API - Fixed possible internal error being returned from \texttt{Studio::Bank::getSampleLoadingState} when called on an unloading bank.
14/04/14 1.03.05 - Studio API patch release

Fixes:

- **LowLevel API** - Added ChannelControl to the C# wrapper to match the C++ API.
- **LowLevel API** - Fixed the definition of `ChannelControl.setDelay` and `ChannelControl.getDelay` in the C# wrapper.
- **LowLevel API** - Replaced broken C# wrapper `System.set3DSpeakerPosition` and `System.get3DSpeakerPosition` functions with `System.setSpeakerPosition` and `System.getSpeakerPosition`.
- **LowLevel API** - Fixed the capitalization of `DSP.setMeteringEnabled`, `DSP.getMeteringEnabled` and `DSP.getMeteringInfo` in the C# wrapper.
- **LowLevel API** - Fixed hang on XMA playback
- **Lowlevel API** - Fix crash when running out of memory loading an ogg vorbis file.
- **Lowlevel API** - Fix symbol collisions when statically linking both FMOD and Xiph libvorbis or libtremor.
08/04/14 1.03.04 - Studio API patch release

Features:

- **Studio API** - [Studio::EventInstance::start](#) now does a full restart of the event if already playing. Restarting events will trigger the [FMOD_STUDIO_EVENT_CALLBACK_RESTARTED](#) callback type.
- **Studio API** - Added [Studio::MixerStrip::setMute](#) and [Studio::MixerStrip::getMute](#) for muting buses.
- **Studio API** - Added [Studio::System::getBufferUsage](#) and [Studio::System::resetBufferUsage](#) for querying command and handle buffer size usage.
- **LowLevel API** - [System::createSound](#) and [System::createStream](#) now faster due to file extension check and immediate prioritization of the relevant codec, before scanning rest of codec types.
- **LowLevel API** - Paused channels now have an effective audibility of 0 and will go virtual if [FMOD_INIT_VOL0_BECOMES_VIRTUAL](#) is enabled.

Fixes:

- **Studio API** - Added [Studio.System.initialize](#) to the C# wrapper to match the C++ API (replacing Studio.System.init, which is now deprecated).
- **Studio API** - Added [Studio.System.create](#) to the C# wrapper to match the C++ API (replacing Studio.Factory.System_Create, which is now deprecated).
- **LowLevel API** - Fix for VBR sounds that dont use [FMOD_CREATECOMPRESSESAMPLE](#) and [FMOD_ACCURATETIME](#) not looping when [FMOD_LOOP_NORMAL](#).
was set.

- **LowLevel API - XboxOne** - Fixed rare mixer hang when playing XMA as a compressed sample.
- **LowLevel API** - Fix crash with combination of `FMOD_OPENUSER + FMOD_NONBLOCKING` and a null pointer being passed to `System::createSound/createStream`.

**Notes:**

- Added examples to the Programmer API documentation.
- **Xbox One** - Now built with March 2014 QFE1 XDK.
- **Studio API** - In the C# wrapper, `Studio.System.init` is now deprecated in favour of `Studio.System.initialize`.
- **Studio API** - In the C# wrapper, `Studio.Factory.System_Create` is now deprecated in favour of `Studio.System.create`.
- **Studio API** - `Studio::EventDescription::is3D` now returns true if any of its nested events are 3D.
- **LowLevel API** - `FMOD_CREATESOUNDEXINFO.suggestedsoundtype` now tries the suggested type first, then tries the rest of the codecs later if that fails, rather than returning `FMOD_ERR_FORMAT`.
- **LowLevel API** - Custom codecs. The open callback for a user created codec plugin now does not have to seek to 0 with the file function pointer before doing a read.
28/03/14 1.03.03 - Studio API patch release

Fixes:

- *Studio API* - Fixed `Studio::EventDescription::getInstanceCount` and `Studio::EventDescription::getInstanceList` incorrectly providing data for all events in the bank, not just the queried event.
- *Studio API* - Added `Studio::EventDescription::loadSampleData`, `Studio::EventDescription::unloadSampleData` and `Studio::EventDescription::getSampleLoadingState` to C# wrapper.
26/03/14 1.03.02 - Studio API patch release

Features:

- *Studio API* - Added Studio::EventDescription::loadSampleData, Studio::EventDescription::unloadSampleData and Studio::EventDescription::getSampleLoadingState functions.
- *LowLevel API* - Added `FMOD_ChannelGroup_IsPlaying` to C API.

Fixes:

- *Studio API* - Fix for setting parameter values that could cause volume changes without the appropriate volume ramp.
- *LowLevel API* - Fix for some incorrect declarations in the C header files.
- *LowLevel API* - Fixed a linker error when calling some C API functions.
- *LowLevel API* - Fixed `FMOD_ChannelGroup_AddGroup` not returning the DSP connection on success.
- *LowLevel API* - *PS4* - Fixed FMOD macros for declaring plugin functions.

Notes:

- *Studio API* - Studio::EventInstance::getLoadingState is now deprecated in favour of Studio::EventDescription::getSampleLoadingState.
18/03/14 1.03.01 - Studio API patch release

Important:

- *LowLevel API* - Blocking commands are not allowed to be called from the low level non-blocking callback. Attempting to do so will log an error and return `FMOD_ERR_INVALID_THREAD`. See `FMOD_SOUND_NONBLOCK_CALLBACK` for more information.

Fixes:

- *Studio API* - Fixed simple nested events not terminating properly when inside multi sounds
- *LowLevel API* - Fix SRS downmix crash on startup if software mixer was set to 5.1, and the OS was set to stereo, and the system sample rate was not 44/48/96khz
- *LowLevel API* - Fix for deadlock that could occur when executing commands in the non-blocking callback as another thread is releasing sounds.
- *LowLevel API* - Fix for `Channel::getPosition` and `ChannelControl::getDSPClock` returning errors when called on emulated channels created with `System::playDSP`.
- *LowLevel API* - *PS4* - Fixed leak of audio output handles on shutdown.
- *LowLevel API* - *PS4* - Fix crash in compressor when placed on a channel with a delay.

Notes:

- *PS4* - Now built with SDK 1.600.071.
03/03/14 1.03.00 - Studio API minor release

Important:

- Added PS Vita platform support.
- Updated FMOD Studio Programmers API documentation.
- Studio API - Changed .bank file format - ALL BANKS MUST BE REBUILT
- Studio API - Studio API is now asynchronous by default, with the processing occurring on a new Studio thread. Asynchronous behaviour can be disabled with the FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE init flag.
- Studio API - Studio API classes are now all referenced as pointers. This reflects a change in the handle system to make it thread-safe, more performant and match the C and low level interface.
- Studio API - Event and mixer strip paths now include a prefix in order to guarantee uniqueness. See Studio::System::lookupID.
- LowLevel API - Low Level is now thread-safe by default. Thread safety can be disabled with the FMOD_INIT_THREAD_UNSAFE init flag.
- LowLevel API - Codecs must set waveformatversion to FMOD_CODEC_WAVEFORMAT_VERSION in the FMOD_CODEC_OPEN_CALLBACK.
- LowLevel API - Removed support for digital CD audio

Features:

- Studio API - The new .bank file format provides improved support for backward and forward compatibility. Future version updates will not generally require banks to be rebuilt.
- Studio API - Added support for events duplicated across banks.
- Studio API - Added support for transition marker and loop region probability.
- Studio API - Added support for sounds on transition timelines.
- Studio API - Added asset enumeration functions: Studio::System::getBankCount, Studio::System::getBankList, Studio::Bank::getEventCount, Studio::Bank::getEventList, Studio::Bank::getMixerStripCount, Studio::Bank::getMixerStripList.
• **Studio API** - Added path retrieval functions: `Studio::System::lookupPath`, `Studio::EventDescription::getPath`, `Studio::MixerStrip::getPath`, `Studio::Bank::getPath`.

• **Studio API** - Bank loading now takes an extra flags argument. It is possible to load banks in non-blocking mode in which case the function will return while the bank is still in the process of loading.

• **Studio API** - Added `Studio::System::setAdvancedSettings`.

• **Studio API** - Added `Studio::System::getCPUUsage`.

• **Studio API** - Studio repositories have improved performance and no longer depend on the standard library map.

• **LowLevel API** - The system callback now includes the error callback type which will be invoked whenever a public FMOD function returns a result which is not `FMOD_OK`.

• **LowLevel API** - Optimize `Sound::getNumSyncPoints` when using large FSB files with many subsounds and many syncpoints.

• **LowLevel API** - Made improvements to virtual voices for DSP graphs using sends, returns, fade points, and sounds with varying peak volumes.

• **LowLevel API** - **PS4** - Added recording support.

• **LowLevel API** - **Xbox One** - Added dll loading support.

• **FSBank API** - Added support for exporting peak volume per sound using the `FSBANK_BUILD_WRITEPEAKVOLUME` flag.

**Fixes:**

• **LowLevel API** - Channels now take fade points into account for virtualisation

• **LowLevel API** - Fixed pops when changing Echo DSP Delay parameter

• **LowLevel API** - **Xbox One** - Removed a CPU spike when first playing a compressed sample XMA.

**Notes:**

• **Studio API** - Replaced `Studio::System::lookupEventID` and `Studio::System::lookupBusID` with `Studio::System::lookupID`.

• **LowLevel API** - The system callback now has an extra userdata argument that matches the userdata specified in `System::setUserData`.

• **LowLevel API** - **Xbox One** - APU allocations are now handled internally for developers using memory callbacks or memory pools.
24/02/14 1.02.13 - Studio API patch release

Fixes:

- *LowLevel API* - Removed stalls when removing a DSP chain from a channel
- *LowLevel API* - Fixed `Channel::getPosition` returning incorrect value for streams with very short loops.
- *LowLevel API* - Fixed rare bug with DSP nodes not being set active in the mixer graph.
- *LowLevel API* - Fixed rare bug with DSP metering not being set.
- *LowLevel API* - Fixed incorrect playback of multi-channel PCM8 data.
- *LowLevel API - PS4* - Fixed issues with calling `ChannelControl::setPosition` on AT9 streams and compressed samples.
- *LowLevel API - PS4* - Fixed audio glitches when using the background music port and the system format is not 7.1
- *LowLevel API - PS4* - Added loading of plugins from PRX files.
- *LowLevel API - Android* - Fixed crash on low quality Vorbis encoded FSBs.
- *LowLevel API - Android* - Fixed one time memory leak on `System::release` when using OpenGL output mode.
- *Studio API* - Fixed playlist instruments occasionally cutting off too early.
- *Studio API* - Fixed rare timing issue that caused spawning instruments to trigger too early.

Notes:

- *Xbox One* - Now built with August QFE11 XDK.
- *PS4* - Now built with SDK 1.600.051.
07/01/14 1.02.12 - Studio API patch release

Fixes:

- LowLevel API - Fixed potential crash with net streams.
- LowLevel API - PS4 - Fixed rare internal error in AT9 codec when channels are reused after stopping.
- Studio API - Fixed nested events getting incorrect 3D position information
17/12/13 1.02.11 - Studio API patch release

Features:

- **LowLevel API** - Added `ChannelControl::setVolumeRamp` and `ChannelControl::getVolumeRamp` to control whether channels automatically ramp their volume changes.
- **Studio API** - Added `FMOD_STUDIO_EVENT_CALLBACK_IDLE` callback type, fired when an event instance enters the idle state.

Fixes:

- **Studio API** - Fixed `FMOD_STUDIO_EVENT_CALLBACK_STOPPED` callback firing when an event instance is already stopped.
- **Studio API** - Fixed `Studio::EventInstance::getCueCount` returning 1 even on events with no sustain points.
- **LowLevel API** - Fixed `ChannelControl::setDelay` rarely being ignored.

Notes:

- **LowLevel API** - PCM data will now be read the main data in a single read instead of breaking the reads up into 16kb chunks.
- **Studio API** - Changed behavior of `Studio::EventInstance::getCue` to return `FMOD_ERR_EVENT_NOTFOUND` if the event has no sustain points.
02/12/13 1.02.10 - Studio API patch release

Important:

- *LowLevel API* - Updated the C ChannelGroup functions to take 64 bit integer argument.

Fixes:

- *LowLevel API* - Fix [FMOD_SPEAKERMODE_SURROUND](#) upmixing to 5.1 or 7.1 incorrectly, ie surround left mixing into LFE and surround right into surround right.
- *LowLevel API - PS4* - Fix playback of background music when system software format is not 7.1.
26/11/13 1.02.09 - Studio API patch release

Notes:

- PS4 - Now built with SDK 1.500.111
19/11/13 1.02.08 - Studio API patch release

Important:

- **LowLevel API** - DSP clock now uses 64 bit integers. The following functions have been modified to accept a 64 bit integer argument: `ChannelControl::getDSPClock`, `ChannelControl::setDelay`, `ChannelControl::getDelay`, `ChannelControl::addFadePoint`, `ChannelControl::removeFadePoints`, `ChannelControl::getFadePoints`.

Features:

- **Studio API** - Added `setParameterValue` and `setParameterValueByIndex` functions in `eventInstance` to wrap finding and then setting a parameter value.

Fixes:

- **LowLevel API** - Fixed positioning of 5.1 surround speakers when soundcard is set to other surround formats
- **LowLevel API** - Fixed excessive log spam making the logging version much slower
- **LowLevel API** - **Xbox One** - Fixed rare XMA codec hang which could also manifest as `FMOD_ERR_INTERNAL`.
- **LowLevel API** - **PS4** - Fixed crash when assigning a channel group to the controller speaker.
- **Studio API** - Fixed MixerStrip release not working when the user has multiple handles to the same strip
- **Studio API** - Fixed pops when playing nested events that have silent tracks
- **Studio API** - Fixed crash when shutting down with profiler connected.
- **Studio API** - Fixed unused streams being created during event preloading

Notes:

- **LowLevel API** - Turned off optimization for user created DSP effects that do not call the read callback if no sound is coming in. read callbacks will now always fire regardless. 'shouldiprocess' callback can be defined to optimize
out no input.
12/11/13 1.02.07 - Studio API patch release

Fixes:

- LowLevel API - iOS - Fixed streams returning `FMOD_ERR_INTERNAL` on ARM64 devices.
- LowLevel API - iOS - Fixed automatic interruption handling not working for ARM64 devices.
- Lowlevel API - Fix possible crash on startup, if using 5.1 mixing on a stereo output (downmixer enabled).
- LowLevel API - Fix setMute on master channelgroup not working.
- Studio API - Fixed AHDSR modulators starting at the wrong value when attack time is 0
- Studio API - Fixed Multi Sounds and Scatterer Sounds not randomizing correctly after deleting all entries
06/11/13 1.02.06 - Studio API patch release

Features:

- LowLevel API - iOS - Added support for ARM64 devices and x86_64 simulator.

Fixes:

- Studio API - Fix playback issues after running for more than 12 hours
- LowLevel API - Fixed net streaming truncating or repeatings parts of the end of a netstream.
- LowLevel API - Fix crash due to missing functions in kernel32.dll on Windows XP.
29/10/13 1.02.05 - Studio API patch release

Features:

- Studio API - Improved performance of `Studio::System::setListenerAttributes`
- Studio API - FMOD profiler can now show Studio Bus and Event instances in the DSP node graph.

Fixes:

- Studio API - Fixed pan jittering on events that move with the listener
Features:

- *Studio API* - Added ability to continue loading banks when missing plugins.
- *Studio API* - Added `FMOD_STUDIO_PARAMETER_TYPE` enum to describe the type of a parameter to `FMOD_STUDIO_PARAMETER_DESCRIPTION`.
- *LowLevel API* - Added function to get parent sound from a subsound.
- *LowLevel API* - *Android* - Added support for dynamic plugins.

Fixes:

- *Studio API* - Trying to set an automatic parameter will return `FMOD_ERR_INVALID_PARAM`.
- *LowLevel API* - Fix restarting a channel corrupting fader and panner positions if effects are added.
- *LowLevel API* - Fix channel restarting if 1. sound ended, 2. `Channel::setVolume(0)` with `FMOD_VOL0BECOMESVIRTUAL` happened, 3. `setVolume(>0)` happened, in between 2 system updates.
- *LowLevel API* - Fixed issues on PS4 after opening an output audio port fails.
- *LowLevel API* - Calling `playSound` on a fsb loaded with `createStream` will now return `FMOD_ERR_SUBSOUND`. 
15/10/13 1.02.03 - Studio API patch release

Fixes:

- *LowLevel API - iOS* - Fixed potential crash when stopping virtual channels.
- *Studio API* - Fixed click with cross-fade for nested events
- *Studio API - Mac* - Fixed link issues from certain API functions.

Notes:

- *Studio API* - [FMOD_Studio_System_Create](#) now takes a headerVersion parameter to match the C++ API
07/10/13 1.02.02 - Studio API patch release

Fixes:

- *LowLevel API* - Fixed rare crash when using virtual voices.
- *LowLevel API* - Fixed channel fade state not being preserved when switching to virtual.
- *LowLevel API* - Fixed 5.1 and 7.1 downmix to stereo being off-center.
- *LowLevel API - Mac* - Fixed incorrect downmix logic causing excess channels to be dropped.

Notes:

- *LowLevel API* - changed *FMOD_DSP_LIMITER_MODE* parameter to *bool*
01/10/13 1.02.01 - Studio API patch release

Features:

- *LowLevel API* - Improved performance of compressor on X86/x64 platforms.
- *Studio API* - Added support for new automatic parameters: Event Orientation, Direction, Elevation and Listener Orientation

Fixes:

- *LowLevel API* - Fixed crash when downmixing to 16-bit output.
- *LowLevel API* - Fixed floating point issue when setting very low pitch values.
- *Studio API* - Fixed sound glitch that could occur after crossfade.
- *Studio API* - Fix for assert when rescheduling with modified pitch.

Notes:

- *iOS* - Now built with SDK 7.0.
- *Mac* - Now built with SDK 10.8.
23/09/13 1.02.00 - Studio API minor release

Important:

- Added Android platform support.

Features:

- **LowLevel API** - Added `FMOD_CREATESOUNDEXINFO.fileuserdata` to hold user data that will be passed into all file callbacks for the sound.
- **LowLevel API** - Added `System::mixerSuspend` and `System::mixerResume` for mobile platforms to allow FMOD to be suspended when interrupted or operating in the background.
- **LowLevel API** - Added float parameter mappings support to plug-ins
- **LowLevel API** - PS4 - Added Output Ports example
- **Studio API** - Reduced memory overhead for several core types.
- **Studio API** - Added `System::System::loadBankMemory` to support loading banks from a memory buffer
- **Studio API** - Added `System::System::loadBankCustom` to support loading banks using bank-specific custom file callbacks
- **Studio API** - Added support for placing multiple tempo markers on a timeline.
- **Studio API** - Better error reporting for instruments scheduled in the past.

Fixes:

- **LowLevel API** - Fix incorrect error codes being returned by C# wrapper.
- **LowLevel API** - Fix bug in stereo-to-surround and surround-to-surround panning
- **LowLevel API** - Fix `System::playDSP` not working
- **LowLevel API** - Fix rare crash with `DSPConnection::setMixMatrix`. Studio API could also be affected.
- **LowLevel API** - Fix `getMeteringInfo` not clearing its values when pausing.
- **LowLevel API** - Fix audio pops when restarting sounds due to downmixing.
- **LowLevel API** - PS4 - Fix crash when disconnecting a channel group from an output port.
- **LowLevel API** - PS4 - Fix issue with audio channels not finishing correctly
when being played through a port.

- **LowLevel API - Xbox One** - Fixed 'clicking' when a realtime decoded XMA sample loops if adjusting pitch during playback.
- **Studio API** - Fix event priority not working
- **Studio API** - Fix **Studio::EventInstance::start()** returning incorrect result with non-blocking sounds.
- **Studio API** - Fix memory leaks when loading corrupt banks
- **Studio API** - Fix channels leaking with nested instruments
- **Studio API** - Fix **MixerStrip::setFaderLevel** on the game side affecting volume levels in the tool when connected via Live Update
- **Studio API** - Fix a potential crash when getting a string property with **Studio::EventDescription::getUserPropertyByIndex**

**Notes:**

- **Studio API** - Renamed **Studio::System::loadBank** to **Studio::System::loadBankFile**
- **Studio API** - Updated the API examples to use the new example project
- **LowLevel API** - Changed **FMOD_FILE_OPEN_CALLBACK** userdata parameter from void** to void* (it now comes from **FMOD_CREATESOUNDEXINFO.fileuserdata** rather than being set by the open callback)
- **LowLevel API - iOS** - Removed automatic handling of interruptions. Developers should call the new **System::mixerSuspend** / **System::mixerResume** API from their interruption handler.
- **LowLevel API - iOS** - Removed all usage of AudioSession API, developers are now encouraged to use the platform native APIs as there is no possible conflict with FMOD.
- **PS4** - Now built with SDK 1.020.041.
02/09/13 1.01.15 - Studio API patch release

Features:

- *LowLevel API* - Performance optimizations.
- *Studio API* - Performance optimizations.

Fixes:

- *LowLevel API* - Fix oscillators not changing pitch if `System::playDSP` was used.
- *LowLevel API* - Fix crash when setting 0 or invalid pitch.
- *Studio API* - Fix for some allocations not propagating `FMOD_ERR_MEMORY` errors.
- *Studio API* - Fix for memory leak when failing to load a bank.
26/08/13 1.01.14 - Studio API patch release

Fixes:

- *LowLevel API* - Fix crash if adding an `FMOD_DSP_TYPE_FADER` dsp to a channelgroup.
- *LowLevel API - Xbox One* - Internal WASAPI (mmdevapi.dll) threads will now have their affinity set to match the FMOD feeder thread.

Notes:

- *Xbox One* - Now built with August XDK.
19/08/13 1.01.13 - Studio API patch release

Features:

- *Studio API* - Global mixer strips will now be automatically cleaned up when when the events routed into them complete.

- *Studio API* - Improved performance of Studio::System::Update by removing stalls waiting on the mixer the complete.

Fixes:

- *LowLevel API* - `Channel::setPitch()` now returns an error if a NaN is passed in. Fixes crashes occurring later in the mixer thread.

- *Studio API* - Fixed sustain points at the start of the timeline not working

- *Studio API* - Fixed sustain point keyoff incorrectly being ignored if the cursor is not currently sustaining

- *Studio API* - Fixed sustain point keyoff incorrectly skipping sustain points repeatedly when looping

Notes:

- *Studio API* - Changed behavior of Studio::EventInstance::getCue to return `FMOD_ERR_INVALID_PARAM` if the event contains no sustain points.
Features:

- Added FMOD SoundBank Generator tool for creating .fsb files. Both a GUI version (fsbank.exe) and a command line version (fsbankcl.exe) are provided.

Fixes:

- *LowLevel API* - Fix FSB Vorbis seek table containing an invalid entry at the end.
- *LowLevel API* - Fix cpu stall when using `System::playDSP`. Also if using oscillator in studio.
- *LowLevel API - Xbox One* - Fixed rare crash on `System::init` when using WASAPI.
05/08/13 1.01.11 - Studio API patch release

Fixes:

- *Studio API* - Fixed resource leak.
- *Studio API* - Fixed a crash when releasing an event instance with sub events.
- *LowLevel API* - Fixed `FMOD_SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED` not being passed to the application.
- *LowLevel API* - *PS4* - Fix crashes caused by out-of-memory conditions. `FMOD_ERR_MEMORY` is now returned correctly.
- *LowLevel API* - *Xbox One* - Fixed race condition that causes a hang when playing compressed XMA samples and streams at the same time.
- *LowLevel API* - *Xbox One* - Fixed leak of SHAPE contexts that would cause `createSound` to fail if playing and releasing lots of XMA streams.
- *LowLevel API* - *Xbox One & Win* - Fixed surrounds and rears being swapped in 7.1.
29/07/13 1.01.10 - Studio API patch release

Important:

- Studio API - Changed .bank file format - API is backward compatible but must be upgraded for compatibility with Studio tool 1.01.10 or newer.

Fixes:

- Studio API - Fixed plugin effect sounds not working in game.
- Studio API - Fixed a crash in Studio::System::update after calling Studio::EventDescription::releaseAllInstances
- Studio API - Fixed sustain points at the start of the timeline not working
- Studio API - Fixed sustain point keyoff incorrectly being ignored if the cursor is not currently sustaining
- Studio API - Fixed sustain point keyoff incorrectly skipping sustain points repeatedly when looping
- Studio API - Fixed a crash when unloading a bank that contains a nested event that is currently playing
- Studio API - Fixed Studio::System::update sometimes failing with FMOD_ERR_INTERNAL and leaving the system in an inconsistent state
- LowLevel API - Xbox One - Fixed FMOD_CREATECOMPRESSESAMPLE XMA playback issues.

Notes:

- Studio API - Changed behavior of Studio::EventInstance::getCue to return FMOD_ERR_INVALID_PARAM if the event contains no sustain points.
22/07/13 1.01.09 - Studio API patch release

Important:

- LowLevel API - Fixed rare crash in mixer.

Features:

- Studio API - Fixed spawning sounds not playing at correct 3D position.
- Studio API - Fixed 40ms of latency getting added for each layer of event sound nesting.
- Lowlevel API - Optimized mixer by about 30% in some configurations.

Fixes:

- LowLevel API - Remove FMOD_CHANNELCONTROL union, used in ChannelControl type callbacks, as it was incorrect and using it as a union would have lead to corruption/crash. A simple opaque FMOD_CHANNELCONTROL type is now used for callbacks, and the user should just cast to the relevant channel or channelgroup type.
- LowLevel API - Fixed fade point interpolation on channels with pitch.
- LowLevel API - Fixed race condition when channels were reused after stopping.
- LowLevel API - Xbox One - Fixed hang for short (2KB) XMA files.
- LowLevel API - Xbox One - Fixed incorrect seek offset for XMA files.
- LowLevel API - PS4 - Added support for AT9 streams with greater than 2 channels.
- Studio API - PS4 - Fixed crash when Handle derived classes went out of scope after the dynamic lib was unloaded

Notes:

- PS4 - Now built with SDK 1.0.
Features:

- **Studio API** - Added `Studio::EventDescription::getMinimumDistance`
- **Studio API** - Added `Studio::EventDescription::isStream`

Fixes:

- **LowLevel API** - Fixed crash / corruption from DSP Fader/Panner objects.
- **LowLevel API** - Fixed mod/s3m/xm/mid playback.
- **Studio API** - Fixed `Studio::EventDescription::isOneshot()` incorrectly returning true for an event that has a loop on the logic track.
- **Linux** - Fixed crash on playback of certain CELT streams.
- **Xbox One** - Fixed potential hangs with compressed XMA samples.
- **Xbox One** - Fixed potential silence if XMA sample rate was not one of 24K, 32K, 44.1K or 48K.
10/07/13 1.01.07 - Studio API patch release

Fixes:

- *LowLevel API* - Fix "Sample Rate Change" tag from passing through 0 rate when EOF was hit on certain MP3 files.
- *LowLevel API* - Fix crash when using [FMOD_CREATECOMPRESSEDSAMPLE](#) introduced in 1.01.06
- *LowLevel API - iOS* - Fixed crash when using DSP Echo.
- *LowLevel API - iOS* - Fixed crash in mixer due to misaligned buffers.
08/07/13 1.01.06 - Studio API patch release

Features:

- *XboxOne* - Officially added support for XMA. Please note this requires the July XDK to avoid a hang.
- *Studio API* - Added `Studio::ParameterInstance::getDescription`
- *Studio API* - Added `EventDescription getParameter, getParameterCount and getParameterByIndex` functions

Fixes:

- Fix Sound userdata being overwritten when `FMOD_SOUND_NONBLOCKCALLBACK` was called for a 2nd or more time.
- *LowLevel API* - Fix rare crash in mixer when releasing a ChannelGroup
- *LowLevel API* - Fix 3D Panner DSPs and Studio Event 3d volumes not being considered by virtual voice system.
- *LowLevel API* - Fixed compressor sounding erratic and unresponsive
- *Studio API* - Fixed clicks when a `Studio::ParameterInstance::setValue` call causes sounds to be cut off

Notes:

- *XboxOne* - Now built with July XDK.
- *LowLevel API* - Changed `FMOD_DSP_TYPE_PAN` `FMOD_DSP_PAN_STEREO_POSITION` parameter to go from -100 to 100.
- *LowLevel API* - Changed `FMOD_DSP_TYPE_COMPRESSOR` `FMOD_DSP_COMPRESSOR_ATTACK` parameter to go from 0.1 to 500ms.
28/06/13 1.01.05 - Studio API patch release

Important:

- **LowLevel API** - Changed .fsb file format - ALL BANKS MUST BE REBUILT
- **Studio API** - Changed .bank file format - ALL BANKS MUST BE REBUILT

Features:

- **Studio API** - Added `Studio::System::unloadAll` function

Fixes:

- **LowLevel API** - PS4 - Improved AT9 decoding performance, fixed issue with when a sound has loop points not aligned to frame size, fixed seamless looping playback glitches.
- **LowLevel API** - Fixed bug that was causing virtual channels to stop prematurely.
- **LowLevel API** - Fixed fade points leaking when channels go virtual.

Notes:

- **Studio API** - Effect data parameter buffers are now 16-byte aligned (128-byte aligned on PS3)
- **Studio API** - Added automatic header version verification to `Studio::System::create`
19/06/13 1.01.04 - Studio API patch release

Fixes:

- *Low Level API* - Fix rare crash with Fader DSP unit.
- *Studio API* - Fixed some effects causing events to not stop correctly
- *Studio API* - Fixed multiple concurrent playbacks of one event sometimes failing with `FMOD_ERR_SUBSOUNDS` returned from `Studio::System::update`

Notes:

- *Studio API* - Changed `Studio::EventInstance::getTimelinePosition` to const
- *PS4* - Now built with SDK 0.990.020.
Features:

- *LowLevel API* - Optimized mixer when pausing or delaying nodes. Should provide significant speed increase for Studio runtime projects.
- *LowLevel API* - Add `FMOD_DSPCONNECTION_TYPE_SEND_SIDECHAIN`.

Fixes:

- *LowLevel API* - Fixed silence in certain DSP configurations.
- *LowLevel API* - Fixed virtual channels not stopping correctly when a parent channel group stops due to an end delay.
- *LowLevel API* - Fixed virtual channels not cleaning up fade points correctly.
- *LowLevel API* - Fixed fade points being ignored when channels go from virtual to non-virtual.
- *Studio API* - Fixed crash when playing a multisound after unloading and reloading it's bank.
- *Studio API* - Implemented `Studio::Bank::loadSampleData`, `Studio::Bank::unloadSampleData` and `Studio::Bank::getSampleLoadingState` (they previously did nothing).
- *Studio API* - Fixed crashes and unexpected behavior with sidechains when they are connected to multiple compressors.
- *Studio API* - Fixed a linker error when calling handle assignment operators.
- *Studio API* - Fixed a crash in the game when adding a sound to an event while connected via Live Update.

Notes:

- *LowLevel API* - Specific parameter description structures like `FMOD_DSP_PARAMETER_DESC_FLOAT` no longer inherit from `FMOD_DSP_PARAMETER_DESC`; instead, `FMOD_DSP_PARAMETER_DESC` includes a union of all the specific structures with floatdesc, intdesc, booldesc and datadesc members. The `FMOD_DSP_INIT_PARAMDESC_xxxx` macros have been updated to reflect this.
• *PS4* - Now built with SDK 0.930.060.
31/05/13 1.01.02 - Studio API patch release

Fixes:

- *Studio API* - Fixed getMixerStrip not returning a valid handle when retrieving a VCA.
30/05/13 1.01.01 - Studio API patch release

Fixes:

- LowLevel API - Fix rare crash in DSPFader.
- Studio API - Studio::EventInstance::getParameter and Studio::EventInstance::getCue now use case-insensitive name comparison

Notes:

- Studio API - Renamed Studio::EventInstance::getNumCues to Studio::EventInstance::getCueCount
Important:

- **Studio API** - Changed .bank file format - ALL BANKS MUST BE REBUILT

Features:

- **PS4** - Added support for mono and stereo AT9 FSBs.
- **LowLevel API** - Optimized mixer by about 10%
- **LowLevel API** - Added 'sends' which is a special type of DSPConnection that does not try and execute the input, the output (return) just consumes what was generated by the input (the send). See `FMOD_DSPCONNECTION_TYPE_SEND`
- **LowLevel API** - Added `DSP::getIdle`. Very useful for seeing if a signal is still running to a DSP unit.
- **LowLevel API** - Added `FadePoint API` - now arbitrary volume ramps can be set anywhere on the timeline. ChannelControl::SetDelay removes ramp in/ramp out in favour of this. See `ChannelControl::addFadePoint/removeFadePoints/getFadePoints`.
- **Studio API** - Added `Studio::EventInstance::setTimelinePosition` and `Studio::EventInstance::getTimelinePosition`
- **Studio API** - Disconnect stopped events from the DSP graph to reduce CPU usage
- **LowLevel API** - removed 'sidechain' API, added `FMOD_DSPCONNECTION_TYPE` which is now a parameter to `DSP::addInput`, `FMOD_DSPCONNECTION_TYPE_STANDARD`, `FMOD_DSPCONNECTION_TYPE_SIDECHAIN`, and `FMOD_DSPCONNECTION_TYPE_SEND` are now supported. `DSPConnection::getType` replaces `DSPConnection::isSideChain`

Fixes:

- **LowLevel API** - Fixed FSB Vorbis not working with encryption key enabled.
- **LowLevel API** - Fixed virtual voices not respecting
ChannelControl::setDelay

- **LowLevel API** - Fixed parameter index validation when getting/setting DSP parameters.
- **LowLevel API** - Fixed reverb not always idling when it should.
- **LowLevel API** - Fixed bug with loop count being incorrectly set to infinite.
- **LowLevel API** - Optimised Echo DSP effect on x86/x64 architectures.
- **LowLevel API** - Fixed `ChannelControl::set3DLevel`, `ChannelControl::set3DSpeakerSpread` and stereo 3d sounds not working.
- **LowLevel API** - Fixed flange effect not updating 'rate' parameter if the rate was set before adding it to a channel or channelgroup or system object.
- **LowLevel API** - *PS4* - Added support for music, voice, personal device and pad speaker routing. See `System::AttachChannelGroupToPort` and `fmodorbis.h`.
- **LowLevel API** - *PS4* - Added dynamic linking option.
- **Studio API** - Fixed stop/release behaviour of event instances containing logic markers.
- **Studio API** - Fixed memory corruption in `Studio::System::release`.
- **Studio API** - Fixed `FMOD_STUDIO_STOP_ALLOWFADEOUT` cutting off delay and reverb.
- **PS4** - Fixed closing the FMOD::System causing platform wide networking to be shutdown even if the system did not initialize it.

**Notes:**

- **XboxOne** - Now built with April XDK.
- **PS4** - Now built with SDK 0.930.
09/05/13 1.00.03 - Studio API patch release

Features:

- *LowLevel API* - Added memory callbacks for DSP plugins

Fixes:

- *LowLevel API* - Fixed true peak calculation in loudness meter
- *LowLevel API* - Fix thread related crash in fader DSP and possibly panner DSP.
- *Studio API* - Fixed automatic angle parameter calculation
12/04/13 1.00.02 - Studio API patch release

Fixes:

- *Studio API* - Fixed snapshots sometimes not working on some properties
- *Studio API* - Fixed VCAs applying fader level twice to controlled buses
09/04/13 1.00.01 - Studio API patch release

**Important:**

- *Studio API* - Changed .bank file format - ALL BANKS MUST BE REBUILT

**Features:**

- *PS4 & XboxOne* - Reduced CPU usage with optimized SSE and AVX functions.

**Fixes:**

- *LowLevel API* - Fix potential crash when stopping and starting sounds quickly and a leak for FMOD_CREATECOMPRESSED codecs which made all sounds go virtual.
- *Studio API* - Fixed a crash when connecting to the game via Live Update
- *Studio API* - Fixed serialization of snapshots with automation

**Notes:**

- *XboxOne* - Now built with March XDK.
25/03/13 1.00.00 - Studio API major release

Important:

- *Studio API* - Changed .bank file format - ALL BANKS MUST BE REBUILT

Features:

- *Mac* - Reduced CPU usage with optimized SSE and AVX functions.
- *XboxOne* - Added ability to set affinity via FMOD_Durango_SetThreadAffinity.
- *PS4* - Added ability to set affinity via FMOD_Orbis_SetThreadAffinity.

Fixes:

- *Studio API* - Fixed `Studio::EventDescription::getLength` return incorrect values
- *Studio API* - Fixed playback glitches when sounds are placed end-to-end on the timeline
- *Studio API* - `Studio::System::lookupEventID` and `Studio::System::lookupBusID` now ignore case
- *Studio API* - Fixed playback of Sound Scatterers with non-zero pitch
- Made return DSPs go idle when there is no input from sends
- Fixed sends sometimes going silent if there are multiple sends to a single return
- Fixed rare hang in mixer when using setDelay with a pitch on the parent

Notes:

- *Studio API* - Replaced `Studio::System::lookupID` with `Studio::System::lookupEventID` and `Studio::System::lookupBusID`.
- FSBank API will now always encode PCM FSBs as PCM16 instead of deciding based on the source file format.
- *PS4* - Now built with SDK 0.920.
25/02/13 0.02.04 - Studio API patch release

Fixes:

- *Studio API* - Studio::System::loadBank now returns `FMOD_ERR_PLUGIN_MISSING` instead of `FMOD_ERR_FILE_BAD` when the bank uses a missing plugin
15/02/13 0.02.03 - Studio API patch release

Features:

- *Studio API* - Added `Studio::System::lookupID()` to look up event IDs from paths (using any string tables present in currently loaded banks).
- *Studio API* - Added `Studio::Bank::unload()` to free loaded bank data.
- *Windows* - Added optimisations to the 64 bit build.

Fixes:

- *Studio API* - Fixed a linker error when calling `Studio::EventDescription::getID`
- Fixed constant `FMOD_ERR_MEMORY` in the TTY and hang if `FMOD_ADVANCEDSETTINGS` is used with DSPBufferPoolSize being set to 0.
- Changed custom DSPs with no shouldiprocess callback to only be processed when their inputs are active.
- Fixed high frequency noise coming from send DSP when channel counts mismatches the return DSP.
- Fixed metering not working via LiveUpdate
- *Windows* - fixed bug in 5.1 mixing in 32bit builds.

Notes:

- *XboxOne* - Now built with January XDK.
- *PS4* - Now built with SDK 0.915.
18/01/13 0.02.02 - Studio API patch release

Important:

- *Studio API* - Changed .bank file format - ALL BANKS MUST BE REBUILT
- *Studio API* - Changed function signature for `Studio::System::initialize`, added `STUDIO_FLAGS` field

Features:

- *Studio API* - Added `FMOD_STUDIO_INIT_LIVEUPDATE` flag to make Live Update optional

Fixes:

- *Studio API* - Fixed an internal error on instantiating a VCA when not all of the mixer strips it controls are loaded

Notes:

- *XboxOne* - Now built with December XDK.
11/01/13 0.02.01 - Studio API patch release

Fixes:

- *Studio API* - Fixed Distance and Angle parameters not being created properly by live update
- Fixed reverb effect generating denorm floats after silence
20/12/12 0.02.00 - Studio API minor release

Features:

- Added Xbox360 support.
- Added iOS support.
- Studio API - Added sub-event instantiation via Studio::EventInstance::createSubEvent
23/11/12 0.01.04 - Patch release

Fixes:

- Fixed a crash when calling Studio::EventInstance::release in a callback fired from Studio::EventInstance::stop with FMOD_STUDIO_STOP_IMMEDIATE
- Fixed a linker error when using Studio::CueInstance::trigger
- Fixed a bug in volume conflict resolver
9/11/12 0.01.03 - Patch release

Fixes:

- Fixed a linker error when using `Studio::EventInstance::setPaused`
29/10/12 0.01.02 - Patch release

Fixes:

- Fixed distortion when the distance between 3D sound and the listener is greater than the maximum attenuation distance.
- Fixed memory leaks when playing a persistent event and triggering sounds via parameter changes.
16/10/12 0.01.00 - Minor release

Features:

- Implemented side chaining for FMOD Compressor

Fixes:

- Studio API - Fixed linker error when calling Studio::EventInstance::isVirtual
- Studio API - Added log message for asset not found error

Notes:

- Second Developer Preview release
28/09/12 0.00.04 - Patch release

**Important:**

- *Studio API* - Changed .bank file format - ALL BANKS MUST BE REBUILT

**Features:**

- Add DSP::addSideChain and FMOD_DSP_STATE::sidechainbuffer to allow a DSP unit to support sidechaining from the output of another DSP.

**Fixes:**

- *Studio API* - Fixed Event::getParameter() and retrieving the name of a parameter via EventParameter::getInfo()
- *Studio API* - Added version checking to bank loading, the runtime will return **FMOD_ERR_FORMAT** when attempting to load an old bank
19/09/12 0.00.03 - Patch release

Fixes:

- Fix panning issue introduced in 5.00.02
- Fix possible crackling noises from mixer optimization in 5.00.02
14/09/12 0.00.02 - Patch release

Important:

- Studio API - Changed .bank file format - ALL BANKS MUST BE REBUILT

Features:

- Optimized mixer to be 20% faster in some cases.
- Studio API - Improved performance of event playback containing mono/stereo tracks

Fixes:

- Studio API - Fixed panning different in game to tool
27/08/12 0.00.00 - Initial release

Notes:

- First Developer Preview release
Firelight Technologies FMOD Studio API
Programmer Topics

The following pages contain information and best practices for how to use FMOD Studio.
Firelight Technologies FMOD Studio API
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 showing 16 bit stereo sample data, 44khz.](image)

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 1/44100th of a second, or 1/44th 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:

- \( ms = \frac{\text{samples} \times 1000}{\text{samplerate}} \)
- \( \text{samples} = \frac{ms \times \text{samplerate}}{1000} \)
- \( \text{samplerate} = \frac{\text{samples} \times 1000}{ms} \)
- \( \text{bytes} = \frac{\text{samples} \times \text{bits} \times \text{channels}}{8} \)
- \( \text{samples} = \frac{\text{bytes} \times 8}{\text{bits} \times \text{channels}} \)

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 realtime than a PCM decompressed wav file does. A streaming sound can only be played once at a time, they cannot be spawned multiple times at once like a sample, due to it only having 1 file handle per stream and 1 ringbuffer to decode into. See `FMOD_CREATESTREAM`.
- "Compressed samples" are an option that allows the user to load a certain compressed file format (such as IMA ADPCM, FADPCM, Vorbis, MP2, MP3, AT9 and XMA formats currently). FADPCM/Vorbis/AT9 are only supported through the .FSB container format), 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`.

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`. 
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.

FMOD will automatically select a channel for the sound to play on, you do not have to manage your own channels.
Sub-mixing and ChannelGroups

Instead of processing or controlling channels individually, channels can be grouped into a 'ChannelGroup'. ChannelGroups allow you to operate on a group of channels at a time, and control the mix graph of the audio.

A ChannelGroup can also be thought of as a 'sub mix', as in the signal chain, this would be the point that the channel signals mix into a single buffer. This buffer can then be processed with a DSP effect (see below) once, rather than once for each channel, saving a lot of CPU time.
2D vs 3D

A 3D sound source is a channel that has a position and a velocity in space. 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 and 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::setMixLevelsOutput, Channel::setMixMatrix or Channel::setPan, whereas these commands on a 3D sound will not have any effect, unless you call Channel::set3DLevel to alter the 2D component of the 3D sound. You can blend a sound between 3D and 2D using this function.
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.
DSP

DSP stands for "Digital Signal Processing", and usually relates to processing raw PCM samples to alter the sound. FMOD provides a suite of DSP effects that can alter the sound in interesting ways to simulate real life or exaggerate a sound. Examples of this are echo, reverb, lowpass filtering, flange and chorus. Effects can easily be added to an FMOD channel, or a sub mix, or ChannelGroup with the `ChannelControl::addDSP` function. You also have the option of writing your own effects with `System::createDSP`. See the [Digital Signal Processing (DSP) Architecture and Usage](#) tutorial for more.
Firelight Technologies FMOD Studio API
Getting Started

Introduction

The FMOD Studio and Low Level API has been designed to be intuitive and flexible. In this section an introduction to using the engine as well as the key issues involved in using it effectively will be explained.

FMOD provides a C++ API and also a C API. They are functional identical, and in fact the C++ and C functions can be mixed interchangeably, with the C++ and C classes being able to be casted back and forth. The following examples only show the C++ version.
Initialization

FMOD Studio API Initialization

When using the Studio API, you can create a FMOD Studio System and then call `Studio::System::initialize`. That function will also initialize the in-built low level FMOD system as well. Here is a simple example:

```c
FMOD_RESULT result;
FMOD::Studio::System* system = NULL;

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

// Initialize FMOD Studio, which will also initialize FMOD Low Level
result = system->initialize(512, FMOD_STUDIO_INIT_NORMAL, FMOD_INIT_NORMAL);
if (result != FMOD_OK)
{
    printf("FMOD error! (%d) %s\n", result, FMOD_ErrorString(result));
    exit(-1);
}
```

FMOD Low Level API Initialization (Do not use this if using FMOD Studio API Initialization)

The FMOD Low Level API can be used without needing to use the FMOD Studio API at all. Using the Low Level API gives access to the fundamental abilities of loading and playing sounds, creating DSP effects, setting up FMOD channel groups, and setting sample-accurate fade points and start/stop times. However, when just using the Low Level API, it will not be possible to load Studio banks or load and play Studio events that sound artists have set up in the Studio tool. To initialize FMOD Low Level directly:

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

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(512, FMOD_INIT_NORMAL, 0);  // Initialize FM
if (result != FMOD_OK) {
    printf("FMOD error! (%d) %s\n", result, FMOD_ErrorString(result))    exit(-1);
}

**Advanced Initialization Settings**

FMOD can be customised with advanced settings by calling
`System::setAdvancedSettings` or `Studio::System::setAdvancedSettings` before initialization. For a description of the typical settings for effective virtual voices, see the [Virtual Voice System](#).
Playing a sound (Low Level API only)

The simplest way to get started, and basic functionality of FMOD Low Level API - is to initialize the FMOD system, load a sound, and play it. All functions execute immediately, so the developer will either fire and forget during their main loop execution, or poll for the sound to finish. Playing a sound does not ‘block’ the application.

To execute a simple playSound

1. Load a sound with System::createSound, using the system object handle as described above. This will return a ‘Sound’ handle. This is your handle to your loaded sound.
2. Play the sound with System::playSound, using the Sound handle returned from Step 1. This will return a ‘Channel’ handle.
3. Let it play in the background, or monitor its status with Channel::isPlaying, using the Channel handle returned from Step 2. A channel handle will also go immediately ‘invalid’ when a sound ends, when calling any relevant Channel based function, so that is another way to know a sound has ended. The error code returned will be FMOD_ERR_INVALID_HANDLE.
Using decompressed samples vs compressed samples vs streams

Decompressed Samples

The default mode for createSound is `FMOD_CREATESAMPLE`, which decompresses the sound into memory. This may be useful for distributing sounds compressed, then decompressing them at runtime to avoid the overhead of decompressing the sounds while playing. This can be expensive on mobile devices depending on the format. Decompressing to PCM uses little CPU during playback, but also uses many times more memory at runtime.

Streams

Loading a sound as a streaming, gives the ability to take a large file, and read/play it in realtime in small chunks at a time, avoiding the need to load the entire file into memory. This is typically reserved for Music / Voice over / dialogue or Long ambience tracks. The user can simply play a sound as a ‘stream’ by adding the `FMOD_CREATESTREAM` flag to the `System::createSound` function, or using the `System::createStream` function. The 2 options equate to the same end behaviour.

Compressed Samples

To play a sound as ‘compressed’, simply add the `FMOD_CREATECOMPRESSEDSAMPLE` flag to the `System::createSound` function

Because compressed samples are more complicated, they have larger contexts to deal with (for example vorbis decode information), so there is a constant per voice overhead (up to a fixed limit) for a playing sound. This allocation is typically incurred at `System::init` time if the user calls `System::setAdvancedSettings` and sets a maxCodecs value, or it could happen the first time a sound is loaded with the `FMOD_CREATECOMPRESSEDSAMPLE` flag. This will not be configured by the user so uses the default of 32 codecs for the allocation.
As an example: the vorbis codec has an overhead of 16kb per voice, so the default of 32 vorbis codecs will consume 512kb of memory. This is adjustable by the user to reduce or increase the default of 32, using the System::setAdvancedSettings function as mentioned. The user would adjust the FMOD_ADVANCEDSETTINGS maxVorbisCodecs value for the vorbis case. Other supported codecs are adjustable as well.

The best cross platform codec to used as a compressed sample is Vorbis (from an FSB file) but if it uses too much CPU for your platform (ie mobile), the FADPCM codec is a good second option. It is less compressed, and uses far less CPU cycles to decode, while giving good quality and 4:1 compression. For PS4 or Xbox One, it is better to use the AT9 and XMA codec format respectively, as the decoding of these formats are handled by separate media chips, taking the load off the CPU.
Update

FMOD should be ticked once per game update. When using FMOD Studio, call Studio::System::update, which internally will also update the Low Level system. If using Low Level directly, instead call System::update.

If FMOD Studio is running in asynchronous mode (the default, unless FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE has been specified), then the Studio::System::update will be extremely quick as it is merely swapping a buffer for the asynchronous execution of that frame's commands.
Shut Down

To shut down FMOD Studio call Studio::System::release. If using the Low Level directly, instead call System::release.
Error Checking

In the FMOD examples, the error codes are checked with a macro that calls into an handling function if an unexpected error occurs. That is the recommended way of calling FMOD Studio API functions. There is also a callback that can be received whenever a public FMOD function has an error. See FMOD_SYSTEM_CALLBACK for more information.
Configuration

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 \texttt{System::init}. For examples of these, see \texttt{Studio::System::getLowLevelSystem}, \texttt{System::setAdvancedSettings}, \texttt{Studio::System::setAdvancedSettings}. 
Avoiding stalls while loading a or releasing a sound

One of the slowest operations is loading a sound. To place a sound load into the background so that it doesn’t affect processing in the main application thread, the user can use the \texttt{FMOD\_NONBLOCKING} flag in \texttt{System::createSound} or \texttt{System::createStream}.

Immediately a sound handle is returned to the user. The status of the sound being loaded can then be checked with \texttt{Sound::getOpenState}. If a function is called on a sound that is still loading (besides getOpenState), it will typically return \texttt{FMOD\_ERR\_NOTREADY}. Wait until the sound is ready to play it. The state would be \texttt{FMOD\_OPENSTATE\_READY}.

To avoid a stall on a streaming sound when trying to free/release it, check that the state is \texttt{FMOD\_OPENSTATE\_READY} before calling \texttt{Sound::release}.
Firelight Technologies FMOD Studio API
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 Studio by using the built in `FMOD::Reverb3D` API.
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 Studio 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 Studio 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 is not applied. It is important to note that by default, 2D sounds share this same physical reverb instance, so to avoid 2D sounds having reverb, use `Channel::setReverbProperties` and set `wet = 0`, or shift the 2D Sounds to a different reverb instance, using the same function (adding a 2nd reverb will incur a small CPU and memory hit).

The interpolation of 3D reverbs 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 reverb as described in the tutorial 'Using multiple reverbs'.
Create a 3D Reverb

We will now create a virtual reverb, using the call `System::createReverb3D`, then set the characteristics of the reverb using `Reverb3D::setProperties`.

```cpp
FMOD::Reverb *reverb;
result = system->createReverb3D(&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 `Reverb3D::set3DAttributes` allows us to set the origin position, as well as the area of coverage using the minimum distance and maximum distance.

```cpp
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`.

```cpp
FMOD_VECTOR listenerpos = { 0.0f, 0.0f, -1.0f }; 
system->set3DListenerAttributes(0, &listenerpos, 0, 0, 0);
```
All done!

This is all that is needed to get virtual 3d reverb zones to work. From this point onwards, based on the listener position, reverb presets should morph into each other if they overlap, and attenuate based on the listener's distance from the 3D reverb sphere's center.
Firelight Technologies FMOD Studio API
3D SOUND

Introduction

This section will introduce you to using 3D sound with FMOD Studio. 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.

For information specific to FMOD Studio Events, see the [Studio 3D Events](#) page.
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. ie.

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

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 inverse

Inverse

This is the default FMOD 3D distance model. All sounds naturally attenuate (fade out) in the real world using an inverse distance attenuation. The flag to set to this mode is FMOD_3D_INVERSEROLLOFF 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 and Linear Squared

These are an alternative distance model that FMOD has introduced. It is supported by adding the FMOD_3D_LINEARROLLOFF or FMOD_3D_LINEARARDSQUAREDROLLOFF 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 or linear squared.
Some global 3D settings

The 3 main configurable settings in FMOD Studio 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 Studio 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.

```cpp
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:

```cpp
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).

```cpp
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.

By default FMOD uses a left-handed coordinate system. If you are using a right-handed coordinate system then FMOD must be initialized by passing `FMOD_INIT_3D_RIGHTHANDED` to `System::init`. In either case FMOD requires that the positive Y axis is up and the positive X axis is right, if your coordinate system uses a different convention then you must rotate your vectors into FMOD’s space before passing them to FMOD.

Note for plugin writers: FMOD always uses a left-handed coordinate system when passing 3D data to plugins. This coordinate system is fixed to use $+X = \text{right}$, $+Y = \text{up}$, $+Z = \text{forward}$. When the system is initialised to use right-handed coordinates FMOD will flip the Z component of vectors before passing them to plugins.
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.

```cpp
do {
    UpdateGame();  // here the game is updated and the sources
    system->set3DListenerAttributes(0, &listener_pos, &listener_vel,
    system->update();  // needed to update 3d engine, once per frame
} 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 `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 Studio 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.

When using the low-level, 3D channels have the following behaviour:

- 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.
- 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.
- 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 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 set the speaker mode for FMOD.
Firelight Technologies FMOD Studio API
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_FILEASYNCCANCELCALLBACK 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, 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(FMODASYNCREADINFO *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 FMODASYNCREADINFO 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 FMODASYNCREADINFO 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.

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 FMOD_ASYNCREADINFO structure when performing a deferred read

The FMOD_ASYNCREADINFO 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 reshuffle the queue to make important reads happen before non important reads.
- Read completion is signalled by simply setting the 'result' code of FMOD_ASYNCREADINFO.
- 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;
} FMOD_ASYNCREADINFO;
```

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 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_CREATESOUNDINDEXINFO::decodebuffersize.
  FMOD_ADVANCEDSETTINGS::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 Studio API
Performance Tutorial

Introduction

Measuring and tweaking performance is an important part of any application and being able to scale FMOD from low power portable devices to the very latest in next gen consoles is key to our design. This guide should give you a solid understanding of how to configure FMOD to fit within your audio budget with specific tips no matter which platform you are targeting.

Before we jump into the details lets first consider how performance is measured in FMOD. The primary metric we use when discussing how expensive something is, is CPU percentage. We can calculate this by measuring the time spent performing an action and comparing it against a known time window, the most common example of this is DSP or mixer performance.

What is the mixer and how is it measured? When we talk about mixer performance we are actually talking about the production of audio samples being sent to the output (usually your speakers). At regular intervals our mixer will produce a buffer of samples which represents a fixed amount of time for playback. We call this the DSP block size and it often defaults to 512 samples, when played back at 48KHz it represents ~10ms of audio.

With a fixed amount of samples being produced regularly, we can now measure how long it takes to produce those samples and receive a percentage. For example, if it took us 5ms of CPU time to produce 10ms of audio, our mixer performance would be 50%. As the CPU time approaches 10ms we risk not delivering the audio in time which results in a audio discontinuity known as stuttering.

What else can be measured?

Another key performance area is update, this operation is called regularly to do runtime housekeeping. Our recommendation is you call update once per render frame which is often 30 or 60 times per second. Using the 30 or 60 FPS (frames per second) known time frame we can now measure CPU time spent performing this action to get percentages.
Armed with the ability to measure performance we now need to identify the things that cost the bulk of the CPU time. The most commonly quoted contributor is voice count, following the logic that playing more sounds will take up more CPU time. Following is a list of the main contributors to the cost of sound playback:

- Decoding compressed audio to PCM.
- Resampling the PCM to the appropriate pitch.
- Applying DSP effects to the sound.
- Mixing the audio with other sounds to produce the final output you hear.

Choosing the correct compression format for the kind of audio you want to play and the platform you want to play it on is a big part of controlling the CPU cost. For recommendations on format choice please consult the performance reference for this platform.
Voice Limiting

Once you've settled on a compression format you need to decide how many sounds of that format you want to be audible at the same time. There are three ways you can use to control the number of sounds playable:

- **System::init**(maxChannels, ...) The maximum number of voices playing at once.
- **System::setSoftwareChannels**(numSoftwareChannels) The maximum number of audible voices.
- **FMOD_ADVANCEDSETTINGS** max???Codec The maximum number of decoders where ??? is the compression format.

For a deep dive into how the virtual voice system works and ways to further control voice count please consult the virtual voices tutorial.

It's often hard to gauge what are good values to use for the above three settings. In rough terms maxChannels should be high enough that you don't hit the cap under normal circumstances, so 256, 512 or even 1024 are reasonable choices. Selecting the values for numSoftwareChannels and maxCodecs will depend on the platform and format used. To help choose these values we have provided some recommendations and benchmarks in the performance reference document for this platform.
Tips and Tricks

With a correctly configured compression format and appropriate voice count you are well on your way to an efficiently configured set up. Next up is a series of tips to consider for your project, not all will be applicable but they should be considered to get the best performance from FMOD.

Sample Rate

There are two sample rates you need to think about when optimizing, the System rate and the source audio rate.

You can control the System sample rate by using `System::setSoftwareFormat(sampleRate, ...)`, which by default is 48KHz. Reducing this can give some big wins in performance because less data is being produced. This setting is a trade off between performance and quality.

To control the source audio rate you can resample using your favorite audio editor or use the sample rate settings when compressing using the FSBank tool or the FSBankLib API. All audio will be sent to a resampler when it is played at runtime, if the source sample rate and the System rate match then the resampler can be essentially skipped saving CPU time. Be aware that this will only happen if there are no pitch / frequency settings applied to the Channel, so this trick is often good for music.

DSP Block Size

As mentioned earlier this represents a fixed amount of samples that are produced regularly to be sent to the speakers. When producing each block of samples there is a fixed amount of overhead, so making the block size larger reduces the overall CPU cost. You can control this setting with `System::setDSPBufferSize(blockLength, ...)`, which often defaults to 512 or 1024 samples depending on the platform.

The trade off with this setting is CPU against mixer granularity, for more information about the implications of changing this setting please consult the API reference for that function.
Channel Count

Controlling how many channels of audio are being played can have a big impact on performance, consider the simple math that 7.1 surround has four times as much data to process compared with stereo. There are a few different places where channel count can be controlled to improve performance.

The source sound channel count should be carefully chosen, often mono sources are best, especially for sound that will be positioned in 3D. Reducing the channel count at the source is an easy win and will also decrease the decoding time for that sound.

Setting the System channel count will control how 3D sounds are panned when they are given a position in the world. You set this channel count by specifying a speaker mode that represents a well known speaker configuration such as 7.1 surround or stereo. To do this use `System::setSoftwareFormat(..., speakerMode, ...)`, the default will match your output device settings.

As a more advanced setting you can limit the number of channels produced by a sub-mix or the number of channels entering a particular DSP effect. This can be especially useful for limiting the channels into an expensive DSP effect. The API to control this is `DSP::setChannelFormat(..., speakerMode)`, by default this will be the output of the previous DSP unit.

DSP Choice

Not all DSPs are created equal, some are computationally simple and use very little CPU, others can be quite expensive. When deciding to use a particular effect it is important to profile on the target hardware to fully understand the CPU implications.

Positioning of the DSP can make a big difference, placing the effect on every voice could cost a lot of CPU time. There are no strict rules for where each effect should be positioned but to give an example, often low and high pass DSP effects can be used per voice efficiently, but reverb will often only have one instance with all voices sending to a sub-mix.
Wrapping Up

Hopefully now you have a good understanding of the options available for optimizing your usage of FMOD. If in doubt about your particular set up, please contact support@fmod.org, we are more than happy to discuss your specific requirements.
Firelight Technologies FMOD Studio API
FMOD Studio DSP Network

Introduction

This section will introduce you to the FMOD Studio advanced DSP system. With this system you can implement custom filters or create complicated signal chains to create high quality and dynamic sounding audio. The FMOD Studio DSP system is an incredibly flexible mixing engine that has an emphasis on quality, flexibility and efficiency, and 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.

- A blue box is an **FMOD::DSP** unit. This unit is represented by the **FMOD::DSP** class in the FMOD low level header.
- A line between the boxes, is an **FMOD::DSPConnection**. This is what links the DSP units together into a network. Each **FMOD::DSPConnection** has a pan matrix which you can use to configure the mapping from input speakers/channels to output speakers/channels.
- The green vertical bars inside the grey bars are detected signal levels. You can see that the WaveTable unit produces a mono signal, that mono signal continues through the Channel Fader (untouched) then gets upmixed to 6 channels (5.1). Because the default pan for a mono sound to a 5.1 output is to have the mono signal attenuated by 3db to the Front Left speaker, and the signal attenuated by 3db to the Front Right speaker, you can see that the 6 grey bars have only signal in the first 2 speaker levels. See **FMOD_SPEAKER** for the speaker order, represented by those bars. **Note:**
Since FMOD Studio 1.04.08, the upmix happens internally, beyond the master ChannelGroup's fader, so for the purposes of this tutorial, the master ChannelGroup's fader has been forced to `FMOD_SPEAKERMODE_5POINT1` so that it can be visualized. More about channel formats can be read below in the "Set the output format of a DSP unit, and control the pan matrix for its output signal" section.

The above image was taken using the FMOD Profiler tool. You can profile your own DSP network as long as you specify `FMOD_INIT_PROFILE_ENABLE` when initializing the low level engine. The tool is located in the /bin directory of the SDK.

**Some common units in a DSP network**

This section will describe the units in more detail, from the origin of the data through to the soundcard, from right to left. The following list describes some of the typical DSP units you will see in a graph.

- **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 usually 48khz by default. (22khz on iOS)
- **DSPCodec Unit**  This unit reads decodes compressed data from an FMOD Codec, and passes it to a built in resampler, and then passes the decompressed result to the output.
- **Channel Fader**  This unit provides a top level unit for a Channel to hold onto, and is a place to insert effects for a Channel. A Channel Fader also controls the volume level of a Channel, for example if the user calls `Channel::setVolume`
- **ChannelGroup Fader**  This unit provides a top level unit for a ChannelGroup to hold onto, and is a place to insert effects for a ChannelGroup. A ChannelGroup Fader also controls the volume level of a Channel, for example if the user calls `Channel::setVolume`

When FMOD plays a PCM sound on a Channel (using `System::playSound`), it creates a small sub-network consisting of a Fader and a Wavetable Unit. This would also happen if playing a stream, even if the source data is compressed.
When FMOD plays a compressed sound on a Channel (MP3/Vorbis/XMA/ADPCM usually, loaded with `FMOD_CREATECOMPRESSEDSAMPLE`), it creates a similar small sub-network consisting of a Fader and a DSPCodec Unit.

When FMOD plays a DSP on a Channel (using `System::playDSP`), it creates a small sub-network consisting of a Fader and a standalone Resampler Unit. The DSP that was specified by the user executed by the resampler as a sub-network to the resampler, and is not visible on the profiler.
Watch a DSP network get built (with code examples)

Start off with nothing, then play some sounds

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.

![Diagram showing a basic signal chain](image)

Note that the network only exists of 1 unit. The Master ChannelGroup's DSP Fader Unit (FMOD_DSP_TYPE_FADER). This unit can be used to control the mix output of the entire mix if desired.

Now we shall play a PCM sound with `System::playSound`.

![Diagram showing the addition of channel faders and wave table](image)

Note that the sub-network of a DSP Fader unit (FMOD_DSP_TYPE_FADER), and a system level DSP WaveTable unit have been attached to the Master ChannelGroup's DSP Fader unit.

Let's play the sound again, resulting in 2 channels being active.

![Diagram showing the active channels](image)

Note now that the new Channel targets the same Master ChannelGroup DSP Fader unit, and when 2 lines merge into 1 unit, a 'mix' happens. This is just a
summing of the 2 signals together.

**Add 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
FMOD::Channel *channel;
FMOD::DSP *dsp_echo;
result = system->playSound(sound, 0, false, &channel);
result = system->createDSPByType(FMOD_DSP_TYPE_ECHO, &dsp_echo);
result = channel->addDSP(0, dsp_echo);
```

The figure below shows the FMOD Echo effect inserted at the 'Channel head' or position 0, as specified with the `Channel::addDSP` command (position = 0). The Channel Fader which used to be the head unit, is now shuffled down to position 1.

If we call `Channel::setDSPIndex`

```cpp
result = channel->setDSPIndex(dsp_echo, 1);
```

We can see below, that the echo has now moved down one, and Channel Fader is back at position 0.

**Create a new ChannelGroup and add our Channel to it**

In this example we shall introduce ChannelGroups which are effectively used as sub-mix buses. We can add an effect to a ChannelGroup and if Channels are assigned to that ChannelGroup, all Channels will be affected by any DSP inserted into a ChannelGroup.
These ChannelGroups can then be nested and manipulated to create hierarchical mixing.

```c
result = system->createChannelGroup("my channelgroup", &channelgroup);
result = channel->setChannelGroup(channelgroup);
```

We can now see the newly created ChannelGroup as a stand-alone DSP ChannelGroup Fader between the channel on the right and the Master ChannelGroup Fader on the left.

**Add an effect to the ChannelGroup**

Adding an effect to a ChannelGroup is the same as adding one to a Channel. Use `ChannelGroup::addDSP`

```c
FMOD::DSP *dsp_lowpass;
result = system->createDSPByType(FMOD_DSP_TYPE_LOWPASS, &dsp_lowpass);
result = channelgroup->addDSP(1, dsp_lowpass);
```

We can now see as before, an effect attached to a ChannelGroup Fader, in position 1, the entirety of the ChannelGroup being symbolized by the box around the 2 units.

**Creating an effect and making all Channels send to it.**

This example demonstrates a more complex, and somewhat typical scenario, in which we create a new effect, and every time a Sound plays on a Channel, we connect the new channel to the effect.

**Important note!** Please don't use this example as a standard way to set up reverb. Simply call `System::setReverbProperties` instead and all connection logic is handled automatically. Note the following logic does not handle what happens when a voice goes virtual and is removed from the graph, only to return later. You would only normally use this logic if you wanted to control the 'wet' mix levels individually for an effect, per channel. Otherwise a simple
`ChannelGroup::addDSP` would suffice.

The first step is to add an effect to the master `ChannelGroup`. We do this by calling `System::createDSPByType` again, and then using the `FMOD::DSP` API to manually add connections.

```cpp
FMOD::DSP *dsp_reverb;
FMOD::DSP *dsp_tail;
FMOD::ChannelGroup *channelgroup_master;
result = system->createDSPByType(FMOD_DSP_TYPE_SFXREVERB, &dsp_reverb);
result = system->getMasterChannelGroup(&channelgroup_master);
result = channelgroup_master->getDSP(FMOD_CHANNELCONTROL_DSP_TAIL, & dsp_tail);
result = dsp_tail->addInput(dsp_reverb);
```

This will result in

```
<table>
<thead>
<tr>
<th>ChannelGroup Fader</th>
<th>ChannelGroup Fader</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOD Reverb</td>
<td></td>
</tr>
</tbody>
</table>
```

Note that the `ChannelGroup` from before is still there. This is what the `Channels` will be playing on. The reason we have a `ChannelGroup` here for this example is to keep the `Channels` executing first in the graph, then the reverb second. This raises a topic called 'order of execution' which you can find more information about below and why it may or may not be important to you.

Also note that the reverb is black. This means it is inactive / disabled. All units are inactive by default, so we have to activate them. You can do this with `DSP::setActive`

```cpp
result = dsp_reverb->setActive(true);
```

```
<table>
<thead>
<tr>
<th>ChannelGroup Fader</th>
<th>ChannelGroup Fader</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOD Reverb</td>
<td></td>
</tr>
</tbody>
</table>
```
Now you can see that the reverb has gone from black/inactive to active.

Now we will play a sound on multiple channels with the following code. The code plays the sound paused, gets its Channel DSP head unit, adds the Channel DSP head unit to the reverb, then unpauases the sound.

```cpp
FMOD::DSP *channel_dsp_head;
result = system->playSound(sound, channelgroup, true, &gChannel[0]);
result = channel->getDSP(FMOD_CHANNELCONTROL_DSP_HEAD, &channel_dsp_head);
result = dsp_reverb->addInput(channel_dsp_head);
result = channel->setPaused(false);
```

Note that calling `Channel::setPaused` internally just calls `DSP::setActive` on the Channel's head DSP unit.

Here is the result

The interesting parts here are that the Channel DSP head units now have 2 outputs per channel, and each set of outputs mix to the user created ChannelGroup first, before being passed as the 'dry' signal to the output. The second set of outputs can be considered the 'wet' path and similarly mix to the reverb unit, before being processed by the reverb processor.
Controlling mix level and pan matrices for DSPConnections

Each connection between an FMOD::DSP unit is represented by an FMOD::DSPConnection object. This is the line between the boxes.

The primary purpose of this object type is to allow the user to control the volume / mix level between 2 processing units, and also to control the speaker / channel mapping between 2 units, so that a signal can be panned, and input signals mapped to any output signal, in any way that is needed.

Lets go back to the example above, but with 1 voice, and change its wet mix from the Channel to the reverb from 1.0 (0db) to 0.0 (-80db)

The code around the playsound would have one difference, and that is that addInput will also take a pointer to the resulting FMOD::DSPConnection object.

```cpp
FMOD::DSP *channel_dsp_head;
FMOD::DSPConnection *dsp_connection;
result = system->playSound(sound, channelgroup, true, &gChannel[0]);
result = channel->getDSP(FMOD_CHANNELCONTROL_DSP_HEAD, &channel_dsp_head);
result = dsp_reverb->addInput(channel_dsp_head, &dsp_connection);
result = channel->setPaused(false);
```

We can then update the volume simply with DSPConnection::setMix

```cpp
result = dsp_connection->setMix(0.0f);
```

You can see there is no signal level in the meter for the reverb, because the only input to it is silent.

**Set the output format of a DSP unit, and control the pan matrix for its output signal**

In this section we will grab the first output from the channel_dsp_head and apply a pan matrix to it, to allow mapping of input signal to any output speaker within
the mix.

The first thing to note, is that the Channel Fader outputs mono to the ChannelGroup Fader. This means there's not much to map from and too here. Any matrix representing this signal will be 1 in and 1 out.

To make it more interesting, we can change the output format of a DSP Unit with `DSP::setChannelFormat`.

```cpp
result = channel_dsp_head->setChannelFormat(0, 0, FMOD_SPEAKER_QUAD);
```

Here is the result

![Diagram](image)

You will notice that the ChannelFader now outputs 4 channels, and this gets propagated through the network. A Quad to 5.1 pan has a different default upmix than mono to 5.1, so you will see that the fronts are now slightly lower on the final ChannelGroup Fader unit, and there is some signal now introduced into the Surround Left and Surround Right speakers. Now we will use some code to do something interesting, we will put the newly quad ChannelFader signal's front 2 channels into the rear 2 speakers of the quad output.

```cpp
FMOD::DSPConnection *channel_dsp_head_output_connection;
float matrix[4][4] =
{    /* FL FR SL SR <- Input signa
     /* row 0 = front left out  */ { 0, 0, 0, 0 },
     /* row 1 = front right out */ { 0, 0, 0, 0 },
     /* row 2 = surround left out */ { 1, 0, 0, 0 },
     /* row 3 = surround right out */ { 0, 1, 0, 0 }
};
result = channel_dsp_head->getOutput(0, 0, &channel_dsp_head_output_connection->setMixMatrix(&matrix[0])
```

result = channel_dsp_head_output_connection->setMixMatrix(&matrix[0])
```
We can now see that the first 2 channels are now silent on the output because they have 0s in the matrix where the first 2 input columns map to the first 2 output columns.

Instead the first 2 input columns have 1s where the rows map to the surround left and surround right output speakers.

**Bypass an effect / disable it.**

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

```cpp
result = dsp_reverb->setBypass(true);
```

This has the benefit of not disabling all input units like `DSP::setActive` with false as the parameter would, and allows the signal to pass through the reverb unit untouched (The reverb process function is not called, saving CPU).

The bypassed reverb is represented as greyed out.

Note that many FMOD effects automatically bypass themselves, saving CPU, after no signal, or silence is detected and the effective 'tail' of the effect has played out.

**Order of execution and pull / no pull traversal**

The order of execution for a DSP graph is from right to left, but also top to
bottom. Units at the top will get executed before units at the bottom.

Sometimes it is undesirable to have a user created effect execute the DSP units for the channel, rather than the ChannelGroup it belongs to. This typically doesn't matter, but one case where it would matter is if the user called Channel::setDelay on the channel or ChannelGroup::setDelay on a parent ChannelGroup, to make the sound delay before starting.

The reverb unit has no concept of the delay because the clock it is delaying against is stored in the ChannelGroup it belongs to.

The result is that the reverb will pull the signal and be audible through the reverb processor, and the dry path will still be silent because it is in a delay state.

The workaround in the above reverb example, is to attach the reverb to the master ChannelGroup after the ChannelGroup the Channels will play on is created, so that the ChannelGroup executes first, and the reverb second.

'Send' vs 'Standard' connection type

A second workaround is to stop the reverb pulling data from its inputs. This can be done by using the FMOD_DSPCONNECTION_TYPE 'type' parameter for DSP::addInput. If FMOD_DSP_CONNECTION_TYPE_SEND is used instead of FMOD_DSP_CONNECTION_TYPE_STANDARD, the inputs are not executed, and all the reverb would do is process whatever is mixed to it from a previous traversal to the inputs.

The delay will then work, but the downside to this method is that if the reverb is first, the signal from the channels will be sent after the reverb has processed. This means it will have to wait until the next mix before it can process that data, therefore 1 mix block of latency is introduced to the reverb.
Firelight Technologies FMOD Studio API
FMOD Studio Plug-in SDK

Introduction

Game studios and third-party developers can augment FMOD Studio's built-in suite of effect and sound modules by creating their own plug-ins. By placing plug-ins in FMOD Studio's plug-ins folder, these can be added to tracks or buses, modulated and automated by game parameters just like built-in effect and sound modules.

This document describes how to create plug-ins and make them available to FMOD Studio and the game. It is recommended you follow along with our examples found in api/lowlvel/examples/plugins as they are fully implemented working effects you can use or base your code on.
Accessing Plug-ins in FMOD Studio

A plug-in must be built as a 32-bit dynamic linked library and placed in the plug-ins folder specified in FMOD Studio’s Preferences dialog under the Plug-ins tab. FMOD Studio scans the folder and all sub-folders both on start-up and when the folder is changed by the user. Studio tries to load any libraries it finds (*.dll on Windows or *.dylib on Mac) and ignores libraries which don't support the API.

Detected plug-in sounds will be available via the track context menu in the Event Editor, whereas detected plug-in effects will show up in the effect deck's Add Effect and Insert Effect context menus. When a plug-in module is added to a track or bus, its panel will be displayed in the effect deck. The panel will be automatically populated with dials, buttons and data drop-zones for each parameter.
Basics

Two versions of the plug-in will usually be required - one for FMOD Studio and one for the game.

Studio will require a 32-bit dll or dylib file if running in Windows or Mac respectively. These will be loaded dynamically in Studio as described in the previous section.

Another version of the plug-in must be compiled for the game's target platform. This may also be a dynamic library but, in most cases, can (or must) be a static library or simply compiled along with the game code. In each case, game code is required to load the plug-in prior to loading the project or object referencing the plug-in.

Building a Plug-in

The fmod_dsp.h header file includes all the necessary type definitions and constants for creating plug-ins including the struct `FMOD_DSP_DESCRIPTION` which defines the plug-in's capabilities and callbacks.

If creating a dynamic library, the library must export FMODGetDSPDescription, e.g.:

```c
extern "C" {
  F_DECLSPEC F_DLLEXPORT FMOD_DSP_DESCRIPTION* F_STDCALL FMODGetDSPDescription;
}
```

Dynamic libraries must be compiled for the same architecture as the host (whether FMOD Studio or the game), so if the game is 64-bit, the game version of the plug-in must be 64-bit otherwise the plug-in should be 32-bit.

A free tool such as Dependency Walker can be used to verify that the library is able to be loaded and the proper symbol is exported. In Windows, the symbol will look like `_FMODGetDSPDescription@0`.

Loading the Plug-in in the Game
The plug-in must be registered using the FMOD Studio or low-level API before the object referencing the plug-in is loaded in the game.

The following functions can be used to register a plug-in if it is statically linked or compiled with the game code:

```cpp
FMOD_RESULT FMOD::Studio::System::registerPlugin(const FMOD_DSP_DESCRIPTION *description, unsigned int *handle);
FMOD_RESULT FMOD::System::registerDSP(const FMOD_DSP_DESCRIPTION *description, unsigned int *handle);
```

If the plug-in library is to be dynamically loaded, a plug-in path can be specified prior to initialising the system using the function:

```cpp
FMOD_RESULT FMOD::System::setPluginPath(const char *path);
```

Any plug-ins in this folder will be automatically registered during initialization. Alternatively, a particular plug-in library can be registered using:

```cpp
FMOD_RESULT FMOD::System::loadPlugin(const char *filename, unsigned int *handle, unsigned int priority = 0);
```

Plug-ins do not normally need to be unregistered, but it is possible with either of the following functions:

```cpp
FMOD_RESULT FMOD::Studio::System::unregisterPlugin(const char* name);
FMOD_RESULT FMOD::System::unloadPlugin(unsigned int handle);
```

In these functions, name refers to the name of the plug-in defined in the plug-ins descriptor and handle refers to handle returned by `FMOD::System::loadPlugin()`.

**Plug-in Types**

There are two main plug-in types:

- Effect Modules
- Sound Modules

Both module types are created in the same way - the difference lies in whether the plug-in processes an audio input.

Effect Modules apply effects to an audio signal, they have an input and an output. Effect Modules can be inserted anywhere in FMOD Studio's signal
routing, whether it be on an Event's track or a mixer bus. Examples of different types of plug-in effects include:

- Effects which have the same input and output channel counts such as EQ, compression, distortion etc...
- Effects which perform up- or down-mixing as part of the processing algorithm such as panning or reverb
- Spatialization and any distance/direction effects which respond to a sound's 3D location in the game such as 3D panning, distance filtering, early reflections or binaural audio
- Side-chaining effects such as compression or audio modulation (e.g. ring modulators)

Sound Modules produce their own sound - they do not have an audio input. Sound modules can be placed on tracks inside Events and can be made to trigger from the timeline, game parameter or within another sound module.
The Plug-in Descriptor

The plug-in descriptor is a struct, `FMOD_DSP_DESCRIPTION`, defined in `fmod_dsp.h`, which describes the capabilities of the plug-in and contains function pointers for all callbacks needed to communicate with FMOD. Data in the descriptor cannot change once the plug-in is loaded. The original struct and its data must stay around until the plug-in is unloaded as data inside this struct is referenced directly within FMOD throughout the lifetime of the plug-in.

The first member, `pluginsdkversion`, must always hold the version number of the plug-in SDK it was compiled with. This version is defined as `FMOD_PLUGIN_SDK_VERSION`. The SDK version is incremented whenever changes to the API occur.

The following two members, `name` and `version`, identify the plug-in. Each plug-in must have a unique name, usually the company name followed by the product name. Version numbers should not be included in the name in order to allow for future migration of saved data across different versions. Names should not change across versions for the same reason. The version number should be incremented whenever any changes to the plug-in have been made.

Here is a code snippet from the FMOD Gain example which shows how to initialize the first five members of `FMOD_DSP_DESCRIPTION`:

```c
FMOD_DSP_DESCRIPTION FMOD_Gain_Desc =
{
    FMOD_PLUGIN_SDK_VERSION,
    "FMOD Gain",     // name
    0x00010000,      // plug-in version
    1,               // number of input buffers to process
    1,               // number of output buffers to process
    ...             // ...
};
```

The other descriptor members will be discussed in the following sections.
Thread Safety

Audio callbacks read, process and shouldiprocess are executed in FMOD's mixer thread whereas all other callbacks are executed in the host's thread (game or Studio UI). It is therefore important to ensure thread safety across parameters and states which are shared between those two types of callbacks.

In the FMOD Gain example, two gains are stored: target gain and current gain. target gain stores the parameter value which is set and queried from the host thread. This value is then assigned to current gain at the start of the audio processing callback and it is current gain that is then applied to the signal. FMOD Gain shows how this method can be used to perform parameter ramping by not directly assigning current gain but interpolating between current gain and target gain over a fixed number of samples so as to minimize audio artefacts during parameter changes.
Plug-in Parameters

Plug-in effect and sound modules can have any number of parameters. Once defined, the number of parameters and each of their properties cannot change. Parameters can be one of four types:

- floating-point
- integer
- boolean (two-state)
- data

Parameters are defined in `FMOD_DSP_DESCRIPTION` as a list of pointers to parameter descriptors, `paramdesc`. The `numparameters` specifies the number of parameters. Each parameter descriptor is of type `FMOD_DSP_PARAMETER_DESC`. As with the plug-in descriptor, parameter descriptors must stay around until the plug-in is unloaded as the data within these descriptors are directly accessed throughout the lifetime of the plug-in.

Common to each parameter type are the members `name` and `units`, as well as `description` which should describe the parameter in a sentence or two. The `type` member will need to be set to one of the four types and either of the `floatdesc`, `intdesc`, `booldesc` or `datadesc` members will need to specified. The different parameter types and their properties are described in more detail the sections below.

Floating-point Parameters

Floating-point parameters have type set to `FMOD_DSP_PARAMETER_TYPE_FLOAT`. They are continuous, singled-valued parameters and their minimum, maximum and default values are defined by the `floatdesc` members `min`, `max` and `defaultval`.

The following units should be used where appropriate:

- "Hz" for frequency or cut-off
- "ms" for duration, time offset or delay
- "st" (semitones) for pitch
- "dB" for gain, threshold or feedback
- "%" for mix, depth, feedback, quality, probability, multiplier or generic 'amount'.
- "Deg" for angle or angular spread

These are preferred over other denominations (such as kHz for cut-off) as they are recognised by Studio therefore allowing values to be displayed in a more readable and consistent manner. Unitless 0-to-1 parameters should be avoided in favour of dB if the parameter describes a gain, % if it describes a multiplier, or a unitless 0-to-10 range is preferred if describing a generic amount.

The `FMOD_DSP_DESCRIPTION` members `setparameterfloat` and `getparameterfloat` will need to point to static functions of type `FMOD_DSP_SETPARAM_FLOAT_CALLBACK` and `FMOD_DSP_GETPARAM_FLOAT_CALLBACK`, respectively, if any floating-point parameters are declared.

These will be displayed as dials in FMOD Studio's effect deck.

**Integer Parameters**

Integer parameters have type set to `FMOD_DSP_PARAMETER_TYPE_INT`. They are discrete, singled-valued parameters and their minimum, maximum and default values are defined by the `intdesc` members `min`, `max` and `defaultval`. The member `goestoinf` describes whether the maximum value represents infinity as maybe used for parameters representing polyphony, count or ratio.

The `FMOD_DSP_DESCRIPTION` members `setparameterint` and `getparameterint` will need to point to static functions of type `FMOD_DSP_SETPARAM_INT_CALLBACK` and `FMOD_DSP_GETPARAM_INT_CALLBACK`, respectively, if any integer parameters are declared.

These will be displayed as dials in FMOD Studio's effect deck.

**Boolean Parameters**

Boolean parameters have type set to `FMOD_DSP_PARAMETER_TYPE_BOOL`. They are discrete, singled-valued parameters and their default value is defined by the `booldesc` member `defaultval`. 
The **FMOD_DSP_DESCRIPTION** members `setparameterbool` and `getparameterbool` will need to point to static functions of type

*FMOD_DSP_SETPARAM_BOOL_CALLBACK* and *FMOD_DSP_GETPARAM_BOOL_CALLBACK*, respectively, if any boolean parameters are declared.

These will be displayed as buttons in FMOD Studio's effect deck.

**Data Parameters**

Data parameters have type set to **FMOD_DSP_PARAMETER_TYPE_DATA**. These parameters can represent any type of data including built-in types which serve a special purpose in FMOD. The `datadesc` member `datatype` specifies the type of data stored in the parameter. Values 0 and above may be used to describe user types whereas negative values are reserved for special types described in the following sections.

The **FMOD_DSP_DESCRIPTION** members `setparameterdata` and `getparameterdata` will need to point to static functions of type

*FMOD_DSP_SETPARAM_DATA_CALLBACK* and *FMOD_DSP_GETPARAM_DATA_CALLBACK*, respectively, if any data parameters with `datatype` 0 and above are declared.

Data parameters with `datatype` 0 and above will be displayed as drop-zones in FMOD Studio's effect deck. You can drag any file containing the data onto the drop-zone to set the parameter's value. Data is stored will be stored with the project just like other parameter types.
Multiple plugins within one file

Typically each plugin only has a single definition. If you want to have multiple definitions from within the one plugin file, you can use a plugin list. An example is shown below.

```c
FMOD_DSP_DESCRIPTION My_Gain_Desc = { .. };  
FMOD_DSP_DESCRIPTION My_Panner_Desc = { .. };  
FMOD_OUTPUT_DESCRIPTION My_Output_Desc = { .. };  

static FMOD_PLUGINLIST My_Plugin_List[] =  
{  
  { FMOD_PLUGINTYPE_DSP, &My_Gain_Desc },  
  { FMOD_PLUGINTYPE_DSP, &My_Panner_Desc },  
  { FMOD_PLUGINTYPE_OUTPUT, &My_Output_Desc },  
  { FMOD_PLUGINTYPE_MAX, NULL }  
};

extern "C"  
{
  F_EXPORT FMOD_PLUGINLIST* F_CALL FMODGetPluginDescriptionList()  
  {  
    return &My_Plugin_List;  
  }  
}

Support for multiple plugins via FMODGetPluginDescriptionList was added in 1.08. If the plugin also implements FMODGetDSPDescription, then older versions of FMOD will load a single DSP effect while newer versions will load all effects.

To load plugins at runtime, call `System::loadPlugin` as normal. The handle returned is for the first definition. `System::getNumNestedPlugins` and `System::getNestedPlugin` can be used to iterate all plugins in the one file.

unsigned int baseHandle;
ERRCHECK(system->loadPlugin("plugin_name.dll", &baseHandle));
int count;
ERRCHECK(system->getNumNestedPlugins(baseHandle, &count));
for (int index=0; index<count; ++index)  
{  
  unsigned int handle;
```
ERRCHECK(system-&gt;getNestedPlugin(baseHandle, index, &handle));
FMOD_PLUGINTYPE type;
ERRCHECK(system-&gt;getPluginInfo(handle, &type, 0, 0, 0));
// we have an output plugin, a dsp plugin, or a codec plugin here
}

The above code also works for plugins with a single definition. In that case, the count is always 1 and System::getNestedPlugin returns the same handle as passed in.
Firelight Technologies FMOD Studio API
Handle System

Introduction

The FMOD Studio and Low Level API returns pointers to types. Some of these types are actually implemented as an underlying handle with the handle data represented to the user as a pointer type. This section explains the underlying representation and lifetime of these objects.
General Information

All FMOD types, whether they are represented internally via pointer or handle, look like a pointer type. No matter the type, a null pointer will never be returned as a valid result, but it is not safe to assume anything else about the pointer value. The user should not assume that the pointer value falls in any particular address range, or that it has any zero bits in the bottom of the pointer value address.

All FMOD types are equivalent for both the C and C++ API. It is possible to cast between the appropriate types by re-interpreting the pointer type directly.
**Low Level Channels**

FMOD Channels are returned to the user as a pointer but actually consist of packed handle data. This allows channels to be re-used safely. When channels are stolen and re-used, then the API will return `FMOD_ERR_INVALID_HANDLE`. Internally FMOD can detect the difference between channels and channel groups because the bit pattern of channels always has 1 in the lowest significant bit, whereas channel groups have 0 in the lowest bit.
Low Level Channel Groups

FMOD Channel Groups are returned to the user directly as a pointer. Once the user destroys a channel group, it is not safe to call FMOD functions with that pointer.
Low Level System

FMOD system object is returned to the user directly as a pointer. Once the user destroys the low level system, is is not safe to call FMOD functions with that pointer.
**Studio Types**

FMOD Studio types are returned to the user as a pointer but actually consist of packed handle data. If the underlying type has been destroyed then the API will return `FMOD_ERR_INVALID_HANDLE`. An example of this would be unloading a Studio::Bank and then referencing a Studio::EventDescription belonging to that bank.
Firelight Technologies FMOD Studio API
Lossy audio formats: quality, multichannel and looping

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>44.025*</td>
</tr>
<tr>
<td>56</td>
<td>48</td>
<td>24</td>
<td>12*</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>40</td>
<td>22.05</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>48</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>56</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>224</td>
<td>112</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>160</td>
<td></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 16kHz.
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.

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>
</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.

The audio data is divided into MPEG frames, each containing 1152 samples

During FMOD's MPEG loop encoding the first frame is primed using data from the last frame

If the first frame begins with silence, the priming process can cause artifacts at the start the loop

**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.
MEMORY MANAGEMENT AND CONSERVATION TUTORIAL

Introduction

This section will give some pointers on how to use and save memory in FMOD Studio 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.

```
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 Studio 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 Studio 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_CREATECOMPRESSESAMPLE 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_CREATECOMPRESSESAMPLE is used with an FMOD_HARDWARE buffer it will generate an FMOD_ERR_NEEDSSOFTWARE error.

**Note!** If you use FMOD_CREATECOMPRESSESAMPLE 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_CREATECOMPRESSEDSAMPLE, 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 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 Studio API
Non-blocking sound creation

Introduction

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 (i.e. System::playSound) until it succeeds. Here is an example of polling the sound until it is ready, then playing it.

```c
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)

or

do {
    if (!channel) {
        result = system->playSound(FMOD_CHANNEL_FREE, sound, if (result != FMOD_ERR_NOTREADY) {
            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 Studio API
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>Metric</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>LFFReference</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.
Converting FMOD Reverb parameters

Reverb parameters have changed in FMOD Studio. The following table shows how to convert FMOD_REVERB_PROPERTIES values from the FMOD Ex set of properties to the FMOD Studio set.

<table>
<thead>
<tr>
<th>Property</th>
<th>Units</th>
<th>Range</th>
<th>Conversion from previous FMOD Ex properties</th>
<th>Clamp after conversion</th>
<th>L</th>
<th>R</th>
<th>T</th>
<th>E</th>
<th>C</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecayTime</td>
<td>ms</td>
<td>100 to 20000</td>
<td>DecayTime * 1000</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>EarlyDelay</td>
<td>ms</td>
<td>0 to 300</td>
<td>EarlyDelay * 1000</td>
<td>No</td>
<td>E</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LateDelay</td>
<td>ms</td>
<td>0 to 100</td>
<td>LateDelay * 1000</td>
<td>No</td>
<td>L</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HFReference</td>
<td>Hz</td>
<td>20 to 20000</td>
<td>HFReference</td>
<td>No</td>
<td>H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HFDecayRatio</td>
<td>%</td>
<td>0 to 100</td>
<td>DecayHFRatio * 100</td>
<td>Yes</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diffusion</td>
<td>%</td>
<td>0 to 100</td>
<td>Diffusion</td>
<td>No</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Density</td>
<td>%</td>
<td>0 to 100</td>
<td>Density</td>
<td>No</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LowShelfFrequency</td>
<td>Hz</td>
<td>20 to 1000</td>
<td>LFReference</td>
<td>No</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LowShelfGain</td>
<td>dB</td>
<td>-48 to 12</td>
<td>RoomLF / 100</td>
<td>Yes</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HighCut</td>
<td>Hz</td>
<td>20 to 20000</td>
<td>IF RoomHF &lt; 0 THEN HFReference / sqrt(1 - HFGain) / HFGain ELSE 20000</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EarlyLateMix</td>
<td>%</td>
<td>0 to 100</td>
<td>IF Reflections &gt; -10000 THEN LateEarlyRatio / (LateEarlyRatio + 1) * 100 ELSE 100</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WetLevel</td>
<td>dB</td>
<td>-80 to 20</td>
<td>10 * log10(EarlyAndLatePower) + Room / 100</td>
<td>Yes</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Intermediate variables used in conversion

\[
\begin{align*}
\text{LateEarlyRatio} & \quad \text{pow}(10, (\text{Reverb} - \text{Reflections}) / 2000) \\
\text{EarlyAndLatePower} & \quad \text{pow}(10, \text{Reflections} / 1000) + \text{pow}(10, \text{Reverb}) \\
\text{HFGain} & \quad \text{pow}(10, \text{RoomHF} / 2000)
\end{align*}
\]
Firelight Technologies FMOD Studio API
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 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
- Querying an instance's reverb properties
- Controlling the wet/dry mix of each reverb instance per channel

In the FMOD Studio UI you would typically allow the sound designer to set up their own reverbs on group buses, and use sends and mixer snapshots to allow the sound designer to control the reverb mix for events.
Setting up the reverbs

Below is an example of setting up four reverb instances. You do not need to explicitly create the extra reverb instance DSP objects - the FMOD Studio 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.

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 supply the 'instance' parameter to set which reverb DSP unit will be used for each preset, whilst calling the `System::setReverbProperties` function.

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

Should you wish to get the current System reverb properties, you must specify the instance number in the 'instance' parameter when calling System::getReverbProperties. In this example we will get the properties for Instance 3.

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

Each channel of the FMOD Studio mixer can set their wet/dry mix for each reverb with `Channel::setReverbProperties`. By default a channel will send to all instances. This example sets the instance 1 send value to linear 0.0 (-80 db) (off).

```
result = channel->setReverbProperties(1, 0.0f);
```

To get the reverb mix level to be full volume again, simply set it to 1 (0db)

```
result = channel->setReverbProperties(1, 1.0f);
```

This system supercedes the now obsolete method of using `FMOD_REVERB_CHANNELPROPERTIES` and flags to specify which instance.
Firelight Technologies FMOD Studio API
Virtual Voice System

Introduction

FMOD Low Level API includes an efficient virtual voice system. FMOD Studio API adds another layer of control on top of that with event polyphony. The following sections describe how best to take advantage of the virtual voice system.
Low Level Virtual Voices

FMOD Low Level includes a virtual voice system. It allows the developer to play hundreds or even thousands of sounds at once, but only have a small number actually producing sound. For example a dungeon may have 200 torches burning on the wall in various places but only the loudest will be really playing. FMOD will dynamically make voices virtual or real depending on real time audibility. A sound which is playing far off or with a low volume will become virtual, but will change to a real voice when it comes closer or becomes louder due to Channel or ChannelGroup API calls.

Audibility Calculation

The virtual voice system automatically takes into account the following when calculating audibility:

- The priority of the channel. See the "Peak Volume" section for more information.
- The underlying sound peak volume if available. See the "Sound Priority" section for more information.
- The volume of the channel and any parent channel group, set via `ChannelControl::setVolume`.
- The volume of any applied fade points, set via `ChannelControl::addFadePoint`.
- Whether the channel is paused or muted.
- The effect of any DSP sends and returns, including the send volume.
- The distance effect for DSP panners.
- Any DSP that exposes `FMOD_DSP_PARAMETER_DATA_TYPE_OVERALLGAIN` will contribute to audibility.

The audibility calculation will take into account the hierarchy of ChannelGroups. If a DSP is added to a parent ChannelGroup that exposes a gain parameter, then that will affect all the audibility of all Channels that are children of that ChannelGroup directly or indirectly.

A Channel can be queried for whether it is virtual with the `Channel::isVirtual`
function. When going virtual, the sound's time will still be ticked and any fade points will still continue to interpolate. Any additional DSPs attached to the channel will be preserved. When the channel becomes real again, it will resume as if it had been playing properly.

**Peak Volume**

Peak volume is available for sounds that are exported via FSBank as long as the "Write peak volume" option is enabled. FMOD Studio tool always enables this flag when exporting banks, so FMOD Studio sounds will always have a peak volume. If the peak volume is not present (such as a loose wav file), then the sound will treated as if it had full volume.

**Sound Priority**

FMOD provides a simple and powerful way of controlling which voices go virtual, by using a channel priority. Channel priority set with `Channel::setPriority` or `Sound::setDefaults`, where a smaller integer value corresponds to a higher (more important) priority. If a sound is a higher priority than another, then it will always take precedence regardless of its volume, distance, or gain calculation. Channels with a high priority will never be stolen by those with a lower priority, ever. The only time a channel with a high priority will go virtual is if other channels with an equal or even higher priority are playing, or if `FMOD_INIT_VOL0_BECOMES_VIRTUAL` has been specified and the sound is effectively silent.

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. Avoid using too many priority levels in a fine-grained way. If a sound has a higher priority it will never be stolen, even if it is very quiet compared to a lower priority sound.

**VOL0 Virtual**

An important part of the virtual voice system is the `FMOD_INIT_VOL0_BECOMES_VIRTUAL` flag. When this flag is enabled, channels will automatically go virtual when their audibility drops below the limit
specified in the **FMOD_ADVANCEDSETTINGS** `vol0virtualvol` field. This is useful to remove sounds which are effectively silent, which is both a performance and quality improvement. Since it is only removing silent sounds, there should be no perceived difference in sound output when enabling this flag.

It is strongly recommended that **FMOD_INIT_VOL0_BECOMES_VIRTUAL** is specified in **System::init** or **Studio::System::initialize**, and that the `vol0virtualvol` field is set to a small non-zero amount, such as 0.001. For users of FMOD Studio API, **System::setAdvancedSettings** can be called by getting the **Studio::System::getLowLevelSystem** after **Studio::System::create** but before **Studio::System::initialize**.

### Software Voices vs Virtual Voices

To set the number of virtual voices FMOD will use, call **System::init** with the number of virtual voices specified in the `maxchannels` parameter. To set the number of software mixed channels available, use **System::setSoftwareChannels**. A further limit is available per codec by using **FMOD_ADVANCEDSETTINGS**.

If the virtual voice limit is hit then Channels will be stolen and start returning **FMOD_ERR_INVALID_HANDLE**. Channels which have had their handle stolen in this way are permanently stopped and will never return.

Assuming the number of playing Channels is below the maximum virtual voice limit, then the channel handle will remain valid, but the Channel may be virtual or real depending on audibility. The maximum number of real playing channels will be the limit set by `setSoftwareChannels`, or the limits of the codecs set with **FMOD_ADVANCEDSETTINGS**.

For typical games, it is reasonable to set the `maxchannels` value of **System::init** to some high value, from a few hundred up to a thousand or more. The number of real software channels is often set lower, at anywhere from 32 to 128. This allows the game to create and keep track of a large number of Channels, but still limit the CPU cost by having a small number actually playing at once.

### Virtual to Real Transition

The way the virtual voice system works is that when sounds become real they resume from their proper place, halfway through the sound. To change this
behaviour, 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 `System::playSound`, the virtual voice sorting needs to be done in `System::update` to process what is really virtual and what isn't.
FMOD Studio API Voice Control

FMOD Studio API provides further ways of limiting playing voices on top of the system provided by the Low Level, by using event polyphony. The sound designer can specify a limit to the number of simultaneously playing instances of an event at once. There are currently two modes for event polyphony - voice stealing on or off.

Event Polyphony with Voice Stealing On

In this mode, once more instances are playing than the limit, then some will become virtual. Whether an event has become virtual can be queried with Studio::EventInstance::isVirtual. A virtual event will mute its master channel group, which will cause any playing Channels to go virtual if FMOD_INIT_VOL0_BECOMES_VIRTUAL has been specified. Event virtualization is determined by the following factors:

- Instance volume set via Studio::EventInstance::setVolume.
- Master track volume set in FMOD Studio, including automation and modulation.
- Attenuation from the event's 3D panner.

An event which is virtual may become real at a later time if the audibility increases compared to the other playing instances.

Event Polyphony with Voice Stealing Off

In this mode, once the instance limit has been met, further instances will not play. Instances can still be created, and Studio::EventInstance::start can be called, but they will not actually play. Querying Studio::EventInstance::getPlaybackState will show that the extra instances are not in the playing state. Once instances fail to play then they will not start at a later time, regardless of what happens to the other instances. In this mode, event audibility has no affect on which instances play, it is simply based on which had Studio::EventInstance::start called first.

Interaction with Low Level Virtual Voice System
FMOD Studio events will ultimately create one or more low level Channel objects to play sound. These Channels can go real or virtual based on the max software channels set at initialization time. Therefore it is possible to have events where `Studio::EventInstance::isVirtual` is false, but some or all of the underlying Channels are virtual due to the software channel limit. The Low Level voice system will correctly take into account the bus set-up, distance attenuation, volume settings, and other DSP effects on Studio buses.

Studio Events can affect the Low Level Virtual Voice selection system with the priority value controlled per-event in the FMOD Studio tool. Any Channels created by an Event will have the priority value exported from the FMOD Studio Tool. Unlike the Low Level API, the Studio tool only exposes 5 different values. This is done deliberately, since priority should not be used in a fine-grained way. Keep in mind that a higher priority voice will never be stolen by a lower priority voice, even if it is very quiet.

Another factor to keep in mind is that Event Priority is not inherited for nested events. It is possible to have a very high priority event that has a bunch of low priority nested events. In that case, the sounds may not play even though ultimately the parent event is set to a high priority.
Low Level Profiler

The FMOD Low Level profiler tool displays the DSP graph, and can be used to quickly see which channels have gone virtual. Consider the Channel Groups Example. If we add `FMOD_INIT_PROFILE_ENABLE` and add a call to `System::setSoftwareChannels` with 5, then we see one of the 6 channels has gone virtual:
Firelight Technologies FMOD Studio API
**SRS Surround sound**

**How to enable SRS 5.1 Surround Sound encoding in FMOD**

FMOD Studio uses SRS downmixing automatically, if the software mixer is 5.1 (FMOD_SPEAKERMODE_5POINT1), and the user's output speaker mode is stereo.
If the speaker mode is not 5.1 in the mixer, the mixer will match its speaker mode to that of the output, so SRS downmixing will never be used. In this case, for 3d sound (Vector based amplitude panning) VBAP will be used between 2 speakers only, and no rear speaker support will be activated from a Prologic/SRS decoder.
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 II®, 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 Studio API
Threads and Thread Safety

Introduction

This section will describe the threads FMOD uses, and the thread safety offered by both the Studio API and the Low Level API.
FMOD Thread Types

Studio Thread

This thread processes all Studio API commands and updates Studio events. It is created during `Studio::System::initialize` by default, unless `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE` is specified as a init flag.

Mixer Thread

This thread is the software mixing thread. This is the thread that does the real work of mixing the DSP graph. It is created at `System::init`.

Stream Thread

This thread is used for decoding streams. It is created the first time a sound is loaded as a stream in `System::createSound` with `FMOD_CREATESTREAM`, or `System::createStream`.

Async Loading Thread

This thread is created the first time a sound is loaded with the `FMOD_NONBLOCKING` flag in `System::createSound`.

File Reading Thread

This thread is used for reading from disk for streams, to then be decoded (decompressed) by the Stream thread. It is created the first time a sound is loaded as a stream in `System::createSound` with `FMOD_CREATESTREAM`, or `System::createStream`. 
Thread Affinity

On some platforms, FMOD thread affinity can be customised. See the platform specific Basic Information page for more information.
**FMOD Callback Types**

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 other threads.
FMOD Thread Safety

FMOD Studio API Thread Safety

By default FMOD Studio API is completely thread safe and all commands will execute on the Studio thread. In the case of functions that return handles to the user, the handle will be valid as soon as the function returns it, and all functions using that handle will be available. Therefore, the fact that the commands have been delayed should be completely transparent to the user.

If `Studio::System::initialize` is called with `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE`, then Studio will not be thread-safe as it assumes all calls will be issued from a single thread. Commands in this mode will be queued up to be processed in the next `Studio::System::update` call. This mode is not recommended except for testing or for users who have set up their own asynchronous command queue already and wish to process all calls on a single thread. See the Studio Thread Overview for further information.

FMOD Low Level API Thread Safety

By default FMOD Low Level API is initialized to be thread safe, which means the API can be called from any game thread at any time. Low Level thread safety can be disabled with the `FMOD_INIT_THREAD_UNSAFE` flag in `System::init` or `Studio::System::initialize`. The overhead of thread safety is that there is a mutex lock around the public API functions and (where possible) some commands are enqueued to be executed the next system update. The cases where it is safe to disable thread safety are:

- The game is using FMOD Studio API exclusively, and never issues Low Level calls itself.
- The game is using FMOD Low Level exclusively, and always from a single thread at once.
- The game is using FMOD Studio API and Low Level at the same time, but FMOD Studio is created with `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE` and the Low Level calls are done in the same thread as the FMOD Studio calls.
Firelight Technologies FMOD Studio API
Spatial Audio

Introduction

Historically audio spatialization (the process of taking an audio file and making it sound "in the world") has been all about positioning sound in speakers arranged on a horizontal plane. This arrangement is often seen in the form of 5.1 or 7.1 surround. With the advancement of VR technology more emphasis has been put on making sound as immersive as the visuals. This is achieved by more advanced processing of the audio signals for the traditional horizontal plane as well as the introduction of height spatialization. This has given the rise of the term "spatial audio" which focuses on this more realistic approach to spatialization.

Within FMOD there are several ways you can achieve a more immersive spatialization experience, depending on your target platform some may or may not apply. The following sections outline a few general approaches with specific implementation details contained within.
Channel based approach

The most traditional way to approach spatialization is by panning signal into virtual speakers, so with the introduction of 7.1.4 (7 horizontal plane speakers, 1 sub-woofer, 4 roof speakers) you can do just this.

- Set your FMOD::System to the appropriate speaker mode by calling `System::setSoftwareFormat(0, FMOD_SPEAKERMODE_7POINT1POINT4, 0)`.
- Select an output mode capable of rendering 7.1.4 content `System::setOutput(FMOD_OUTPUTTYPE_WINSONIC)`.
- Specify a buffer size compatible with Windows Sonic `System::setDSPBufferSize(480, 4)`. *NOTE:* This requirement will be removed in a future version.

You can now `System::createSound` and `System::playSound` content authored as 7.1.4. If you have the necessary sound system setup (i.e. Dolby Atmos) you will hear the sound play back including the ceiling speakers. If you have a headphone based setup (i.e. Windows Sonic for Headphones or Dolby Atmos for Headphones) you will hear an approximation of ceiling speakers.

To take an existing horizontal plane signal and push it into the ceiling plane you can create an FMOD spatializer and adjust the height controls.

- Create the spatializer with `System::createDSPByType(FMOD_DSP_TYPE_PAN)`.
- Add it to an FMOD::Channel or FMOD::ChannelGroup with `ChannelControl::addDSP`.
- Control the height by setting `FMOD_DSP_PAN_2D_HEIGHT_BLEND` via `DSP::setParameterFloat`.

Not only will this let you blend to the 0.0.4 ceiling speakers by setting the value between 0.0 and 1.0, it will also let you blend from the 0.0.4 ceiling speakers to the ground plane 7.1.0 by setting the value between 0.0 and -1.0.
Object based approach

To get more discrete spatialization of an audio signal you can use the FMOD object spatializer, so named because the audio signal is packaged with the spatialization information (position, orientation, etc) and sent to an object mixer. Often used to highlight important sounds with strong localization to add interest to a scene, usually used in-conjunction with the channel based approach, be that 7.1.4 or even simply 5.1 / 7.1.

- Set your FMOD::System to an object ready output plugin by calling `System::setOutput(FMOD_OUTPUTTYPE_WINSONIC)` or `System::setOutput(FMOD_OUTPUTTYPE_AUDIO3D)`.
- Create an object spatializer with `System::createDSPByType(FMOD_DSP_TYPE_OBJECTPAN)`.
- Provide 3D position information with `FMOD_DSP_OBJECTPAN_3D_POSITION` via `DSP::setParameterData`.

There is no limit to how many `FMOD_DSP_TYPE_OBJECTPAN` DSPs you create, however there is a flexible limit on a per-platform basis for how many can be processed. When there are more object spatializers in use than there is available resources FMOD will virtualize the least significant sounds by processing with a tradition channel based mix.

An important consideration when using object spatializers is signal flow, unlike most DSPs, after the signal enters the DSP it is then sent out to the object mixer. The object mixer could be a software library or a physical piece of hardware, in all cases though you no longer have access to that signal. Any processing you would like to perform must be done before that point. However (to assist mixing) the object spatializer will automatically apply any "downstream" ChannelGroup volume settings so it behaves similarly to the standard FMOD spatializer.
Third party plugins

In addition to the built-in channel and object based approaches there are third party plugins available that can assist too. The FMOD DSP plugin API (see FMOD_DSP_DESCRIPTION) allows any developer to produce an interface for their spatial audio technology and provide it across all FMOD platforms. Additionally the FMOD output plugin API (see FMOD_OUTPUT_DESCRIPTION) allows developers to implement a renderer for the FMOD object spatializer extending the functionality to more platforms and more technologies.

Resonance Audio Spatializer

Once such third party is the Resonance Audio cross-platform suite of plugins that comes bundled with FMOD. Resonance Audio offers a "Source" plugin which behaves much like the FMOD object spatializer in that audio is sent out to an object mixer, however the final signal returns as binaural output at the "Listener" plugin. Resonance Audio also offers a "Soundfield" plugin for playing back first order Ambisonic sound fields. For more details about the usage of Resonance Audio please check out the user guide.

Oculus Spatializer

Another cross-platform suite of spatial audio plugins is that offered by Oculus as part of their Audio SDK. You can find instructions and downloads for these available on their website.

Coming Soon

Additional plugins available in the future include:

- dearVR
- Steam Audio
- G'Audio Craft
- OSSIC
- RealSpace3D
Usage in FMOD Studio

All of the functionality above has been presented in the context of the lowlevel FMOD API, however it is also all available (including the plugins) within FMOD Studio. For more details about the individual components and their visual representations please read the FMOD Studio manual, for a quick reference of where each feature applies see below.

- 7.1.4 output: Window -> Mixer, select "Master Bus", right click "out" on the deck, set Surround 7.1.4.
- Height control: Use the "height" slider that is part of the deck panner on any bus configured as 7.1.4.
- Object spatialization: Right click the deck for any event, Add effect -> FMOD Object Spatializer.
- Resonance Audio spatialization: Right click the deck for any event, Add effect -> Plug-in effects -> Google -> Resonance Audio Source.

Note: When using Windows Sonic output you must first be running Windows 10 Creators Update. You must also configure it for your audio device. Right click the speaker icon in the system tray -> Playback devices -> Right click your default device -> Properties -> Spatial sound -> Spatial sound format, now choose your desired spatial technology. FMOD will use your default output device with the technology you select here.
Firelight Technologies FMOD Studio API
Transitioning between FMOD Ex and FMOD Studio

Introduction

This section describes the differences between FMOD Ex and FMOD Studio.
FMOD Studio API Versus FMOD Designer API

The FMOD Studio API is conceptually similar to the old FMOD Designer API, but most classes have been renamed, and some have been removed or split up.

EventSystem and EventSystem_Create

The Studio::System class is analogous to the old EventSystem class. The static Studio::System::create function is analogous to the old EventSystem_Create function.

EventProject

The Studio::Bank class is analogous to the old EventProject class.

FEV and FSB Files

Each FMOD Designer project produced a single .fev file and multiple .fsb files. In contrast, an FMOD Studio project produces multiple .bank files, which contain event metadata as well as sample data. See the Studio::System::loadBankFile, Studio::Bank::loadSampleData, and Studio::EventDescription::loadSampleData functions. The low level API still supports loading .fsb files directly.

EventGroup

FMOD Studio doesn't have the concept of EventGroups. Events can be placed in folders, but this has no significance at runtime. To achieve the loading control formerly provided by EventGroups, events can be placed in different banks.

Event

The old Event class has been split into two classes: Studio::EventDescription and Studio::EventInstance. Studio::EventDescription holds the static data that describes an event, while Studio::EventInstance is a playable instance of an event. These two classes correspond to old Event objects retrieved by calling
getEvent with or without the FMOD_EVENT_INFOONLY flag.

The FMOD Designer API used to create a fixed number of event instances when you first called getEvent without FMOD_EVENT_INFOONLY, whereas the FMOD Studio API creates a single instance each time you call EventDescription::createInstance. This gives you more control over the memory usage for each event.

Retrieving Events

Events can be retrieved by ID (using Studio::System::getEventByID) or by path (using Studio::System::getEvent).

Event IDs are GUIDs (globally unique identifiers), and stay the same even if the event is renamed or moved around in the project. IDs can be parsed from a string using Studio::parseID, or looked up from a path using Studio::System::lookupID if the "Master Bank.strings.bank" file is loaded.

Event paths consist of the string "event:/", followed by the the path to the event within the project's folder structure. To retrieve events by path, the "Master Bank.strings.bank" file must be loaded.

EventParameter

The Studio::ParameterInstance class is analogous to the old EventParameter class.

Sustain Points and Key Off

The old EventParameter class had a keyOff function to move past the next sustain point. In FMOD Studio this behaviour is implemented via a built-in KeyOff cue that can be triggered via Studio::EventInstance::triggerCue.

EventCategory

Event categories have been replaced in FMOD Studio by buses and VCAs, which provide much more functionality to the sound designer. At runtime, buses and VCAs are accessed via the Studio::Bus and Studio::VCA classes.
**EventReverb**

The EventReverb class has been removed. Similar functionality can be implemented using events with automatic distance parameters controlling reverb snapshots.

**EventQueue and EventQueueEntry**

FMOD Studio doesn't have the concept of event queues.

**MusicSystem and MusicPrompt**

Interactive music in FMOD Studio is implemented inside the event editor, so the MusicSystem and MusicPrompt classes have been removed.

**NetEventSystem**

Whereas the old network tweaking API was implemented as a separate fmod_event_net library, the FMOD Studio live update system is built into the main API library, and can be enabled by passing `FMOD_STUDIO_INIT_LIVEUPDATE` to `Studio::System::initialize`. 
FMOD Studio Low Level API Versus FMOD Ex Low Level API

New features

No more 'recommended startup sequence'

FMOD Studio now handles the speaker mode automatically. To start FMOD Studio is as simple as calling `System::init`, rather than querying caps or speaker modes to make FMOD's software mixer match that of the operating system. If you want the software mixer to be forced into a certain speaker mode, disregarding the operating system, use `System::setSoftwareFormat`. FMOD will automatically downmix or upmix this mode if the user's system is not the same speaker mode as the one selected for the software engine. Where possible, SRS, Dolby or internal downmixing will be used to achieve better surround effect.

Added new DSP effects

The following new DSP effects are added. (From fmod_dsp_effects.h)

```c
FMOD_DSP_TYPE_SEND, /* This unit sends a copy of the signal to a return DSP anywhere in the DSP tree. */
FMOD_DSP_TYPE_RETURN, /* This unit receives signals from a number of send DSPs. */
FMOD_DSP_TYPE_HIGHPASS_SIMPLE, /* This unit filters sound using a simple highpass with no resonance, but has flexible cutoff and is fast. Deprecated and will be removed in a future release (see FMOD_DSP_TYPE_PAN). */
FMOD_DSP_TYPE_THREE_EQ, /* This unit is a three-band equalizer. */
FMOD_DSP_TYPE_FFT, /* This unit simply analyzes the signal and provides spectrum information back through `getParameter`. */
FMOD_DSP_TYPE_LOUDNESS_METER, /* This unit analyzes the loudness and true peak of the signal. */
FMOD_DSP_TYPE_ENVELOPEFOLLOWER, /* This unit tracks the envelope of the input/sidechain signal. Format to be publicly disclosed soon. */
FMOD_DSP_TYPE_CONVOLUTIONREVERB, /* This unit implements convolution reverb. */
FMOD_DSP_TYPE_CHANNELMIX, /* This unit provides per signal channel gain, and output channel mapping to allow a multichannel signal made up of many groups of signals to map to a single output signal. */
FMOD_DSP_TYPE_TRANSCEIVER, /* This unit 'sends' and 'receives' from a selection of up to 32 different slots. It is ... om a single channel, or multiple transceivers can send to a single channel, or a combination of both. */
FMOD_DSP_TYPE_OBJECTPAN, /* This unit sends the signal to a 3d object encoder like Dolby Atmos. Supports a subset of the */
FMOD_DSP_TYPE_MULTIBAND_EQ, /* This unit is a flexible five band parametric equalizer. */
```

Note FMOD's convolution reverb also has the option to be GPU accelerated for big performance wins.

New file format support
**FMOD_SOUND_TYPE_FADPCM** is a new highly optimized, branchless ADPCM variant which has higher quality than IMA ADPCM, and better performance. Great for mobiles.

**Dolby Atmos support**

Through the FMOD object panner, sounds can be positioned spherically in 3D space to support Dolby Atmos. This includes height speakers.

**New DSP mixing engine**

FMOD Studio's software mixing engine is now more flexible, with support for changing channel formats and speaker modes as the signal flows through the mix. FMOD Studio's new software mixing engine also has better custom DSP support, with branch idling, silence detection, sends and returns, side-chaining, circular connections and more.

**Added** **System::getDSPInfoByPlugin**

This function has been added to plug the gap between loading a plugin, and getting its **FMOD_DSP_DESCRIPTION** structure so that it can be passed to **System::createDSP**.

**Added** **System::mixerSuspend** and **System::mixerResume**

For mobile devices, these functions allow you to halt FMOD's cpu mixer and thread before sleeping or answering a call or any other interruption event.

**ChannelGroups volume and panning are now DSP based, rather than reaching down to the channels of the child channel groups.**

ChannelGroups in FMOD Ex were a way to scale and modify the pan and volume of the channels in the channelgroups below. FMOD Studio now modifies pan and volume on a DSP basis using the FMOD DSP 'Fader' unit, and the fader can be repositioned in amongst its own effect list, so you can get the concept of post or pre fader effects.

**Added** **System::attachChannelGroupToPort** and
**System::detachChannelGroupFromPort**

This new feature allows signals to be sent to alternate sound outputs like headsets, controllers with speakers, and console based multi speaker arrays that are not the standard surround sound speaker output.

**Added Sound::getSubSoundParent**

Added the ability for a child sound (subsound) to query for its parent.

**Added ChannelControl::setVolumeRamp/ChannelControl::getVolumeRamp**

When queuing sounds up and for other reasons, it can be desirable to turn off ramping of volume changes, so the ability has been added to turn volume ramping (declicking) off or on per channel/channelgroup.

**Added UTF8 support**

Functions such as System::getDriverInfoW have been removed in favour of UTF8 support.

**Added ChannelControl::addFadePoint, ChannelControl::removeFadePoints and ChannelControl::getFadePoints**

Now the ability to set up a volume fade or ramp between any 2 clock points is possible. Link up multiple fade points to create envelopes. All fading is clock accurate and compensated for when pitch is changed.

**Added concept of a 'fader' DSP per Channel and ChannelGroup**

Each Channel and ChannelGroup has a built in 'fader' (FMOD_DSP_TYPE_FADER). This also acts as the DSP 'head' of the Channel or ChannelGroup by default. This can of course be changed later. If you add an effect, you can position it to be the head, and the fader will be before that in the signal chain.
A fader DSP simply adjusts volume of the signal for a mix, and also supports sample accurate volume fade points, for envelopes and fade ins / fade outs. A panner is a separate entity which can pan a sound in mono, stereo and surround using a variety of parameters including position/direction/extent/rotation/axis control and 3d position control. The order of processing of a fader and panner can be controlled using `ChannelGroup::setDSPIndex`, and it can be positioned into an effect chain so that panning comes before an effect or after, with fading coming at a different position in the effect chain.

**New DSPConnection types supported**

`DSP::addInput` now has a new optional parameter to describe the connection's behaviour between 2 DSP units. `FMOD_DSPCONNECTION_TYPE` has been added to tell an input DSP that

- It should mix to a 'sidechain' buffer, which is a special buffer for DSP units that want to process a sidechain (for example a compressor) through a DSP unit's `FMOD_DSP_STATE` structure. This signal is not audible but is used to affect behaviour within the DSP. `FMOD_DSPCONNECTION_TYPE_SIDECHAIN` would be used here.
- It should only consume existing data, not execute the input to generate that data. `FMOD_DSPCONNECTION_TYPE_SEND` would be used here. This can be useful for graph dependency reasons, so that an input is not executed before it should be.

**New DSP parameter types**

In FMOD Ex, only float parameters were supported. Now Int, Bool and Data parameters are supported.

Use:

- `DSP::setParameterFloat`, `DSP::getParameterFloat`
- `DSP::setParameterInt`, `DSP::getParameterInt`
- `DSP::setParameterBool`, `DSP::getParameterBool`
- `DSP::setParameterData`, `DSP::getParameterData`
Changes

**FMOD_HARDWARE support**

FMOD Studio is such an advanced mixing engine, that the limited features of hardware voices could not possibly hope to achieve parity with the features of FMOD's software engine. It has been decided that in this day and age of multicore processors, even at the mobile level, that hardware voice support gained nothing for the user. FMOD's software engine can actually be faster, and better quality, and more flexible than hardware, so all sounds and voices are software mixed now. API changes:

- System::setHardwareChannels is removed
- System::getDriverCaps is removed. systemrate, speakermode, speakermodechannels parameters have been added to System::getDriverInfo to provide the information which may be important to the developer. Similarly, System::getRecordDriverCaps has been removed, in favour of System::getRecordDriverInfo.
- System::setSpeakerMode and System::getSpeakerMode are removed, and instead are now moved into System::setSoftwareFormat because 'speakermode' now only affects the internal mixer, and the operating system speaker mode is matched at runtime through matrix upmixing or downmixing (automatically).

**FMOD Ex had Channels and ChannelGroups, now it seems they are merged into 'ChannelControl'?**

Channels and ChannelGroups had some similar functionality in FMOD Ex (ie setVolume, setPaused), but FMOD Studio takes it even further. FMOD Studio has been designed so that ChannelGroups can have a lot more control, like pausing and muting at the DSP node level, adding effects, and positioning in 3d.

**Converting FMOD Reverb parameters**

Reverb parameters have changed in FMOD Studio. See the Reverb notes section for more information.
System::set3DSpeakerPosition is now called System::setSpeakerPosition

Minor name change to denote that 3d and 2d positioning is affected by speaker location.

FMOD_CHANNEL_FREE / FMOD_CHANNEL_REUSE removed. Default ChannelGroup now passed to playSound if needed

System::playSound and System::playDSP now do not take a 'channel' parameter. All channels behave the same way as FMOD_CHANNEL_FREE would have in FMOD Ex.
Seeing as a parameter was removed from playSound/playDSP, a new one was added to remove the overhead of disconnecting and reconnecting DSP graph nodes after playSound was called, so that they could be hooked up to a new ChannelGroup other than the master ChannelGroup. Consider this addition a performance optimization, and a simplification in API calls.

CDROM / CDDA support removed

Legacy support for CDROM redbook audio playback has been removed due to lack of interest and issues with maintaining said code.

System::getSpectrum and System::getWaveData removed

Add a custom DSP unit to capture DSP wavedata from the output stage. Use the master channelgroup's DSP head with System::getMasterChannelGroup and ChannelControl::getDSP.
Add a built in FFT DSP unit type to capture spectrum data from the output stage. Create a built in FFT unit with System::createDSPByType and FMOD_DSP_TYPE_FFT, then add the effect to the master ChannelGroup with ChannelGroup::addDSP. Use DSP::getParameterData to get the raw spectrum data or use DSP::getParameterFloat to get the dominant frequency from the signal.

'W' function wide char support removed
Functions such as System::getDriverInfoW have been removed in favour of UTF8 support.

**System::setReverbAmbientProperties removed**

The 'background' ambient reverb for a 3d reverb system is now just the standard reverb set with System::setReverbProperties.

**System::getDSPClock removed**

Use the DSP clock of the master ChannelGroup instead with System::getMasterChannelGroup and ChannelGroup::getDSPClock.

**getMemoryInfo functions have been removed from all classes**

Due to the unreliability of the function in FMOD Ex due to caching, threads, and shared memory throwing results out, the function has been removed. Alternate memory tracking methods will be added later. FMOD::Memory_GetStats and logging is the best way to track memory at the moment.

**Sound::setVariations and Sound::getVariations have been removed**

Due to this being a 'helper' function that can easily be acheived in user code, this was deemed not necessary and removed.

**Sound::setSubSoundSentence has been removed**

This function has been removed in favour of the extremely precise and more reliable ChannelControl::setDelay functionality. 2 or more sounds can be queued up to play end to end using this function, with the added benefit of cross fades and overlaps.

**setDelay and clock functions now use 64bit 'long long' type rather than than 32bit hi and 32bit low part parameters.**

Due to the complexity of working in fixed point, and seeing as there is a consistent 64bit type for all compilers in long long that FMOD supports, we
have switched to this type. 1 32bit value would wrap around in 24 hours at 48khz, so 64bit values will last for 12 million years which should be enough to avoid in game clock wrap around.

**Channel::setSpeakerMix, Channel::setSpeakerLevels, Channel::setInputChannelMix, Channel::getPan, Channel::getSpeakerMix and Channel::getSpeakerLevels removed**

A cleaner replacement has been made for these functions,

- `ChannelControl::setMixLevelsOutput` (float frontleft, float frontright, float center, float lfe, float surroundleft, float surroundright, float backleft, float backright);
- `ChannelControl::setMixLevelsInput` (float *levels, int numlevels);
- `ChannelControl::setMixMatrix` (float *matrix, int outchannels, int inchannels, int inchannel_hop = 0);
- `ChannelControl::getMixMatrix` (float *matrix, int *outchannels, int *inchannels, int inchannel_hop = 0);

Note that because setPan, setMixLevelsOutput and setMixLevelsInput all affect a final pan matrix, only getMixMatrix is available as the 'getter'. There is no more exclusive mode for the panning technique like there was in FMOD Ex. All 3 functions can now affect the final matrix. The setMixMatrix function now allows instant setting of a full matrix for performance reasons, rather than calling setSpeakerLevels each time for each speaker.

**DSPConnection::setLevels/getLevels have been removed.**

As above, new functionality is available to produce the same result. Rather than setting output levels row by row, a full 2 dimensional matrix is supplied with `DSPConnection::setMixMatrix`.

**Channel::set3DPanLevel misnomer renamed to ChannelControl::set3DLevel**

The FMOD Ex function did not actually only affect pan, it also affected doppler
and distance attenuation so to call it 'pan' level was not really correct. FMOD Studio corrects this.

'override' functions removed from ChannelGroup class

Because volume/pitch/muting/reverb properties and pausing has been added as a ChannelGroup concept at the DSP level, ChannelGroups no longer rely on their 'channels' to do pausing and muting type logic. The ChannelGroup's DSP itself will directly mute/pause etc, thanks to the removal of FMOD_HARDWARE support.

System::addDSP removed from the System API in favor of adding the DSP to the 'master channel group' with System::getMasterChannelGroup and ChannelGroup::addDSP

Rather than using the 'system' to add a DSP to, the user must use System::getMasterChannelGroup which gets the top level channel group, then use ChannelGroup::addDSP to add the DSP to it, which effectively puts at the end of the signal chain.

DSP::remove removed from the DSP API in favourite of ChannelControl::removeDSP

Effects that are added with ChannelControl::addDSP are owned by the ChannelControl that added it, and therefore must be removed with ChannelControl::removeDSP, instead of having the DSP removing itself. This helps the channelgroup or channel manage resources.
Firelight Technologies FMOD Studio API
Studio 3D Events

Introduction

This section will introduce you to using 3D sound with FMOD Studio events.
Coordinate systems and handedness

FMOD Studio shares the same coordinate system as the lowlevel API. See 3D Sound for details.
### Updating orientations

The programmer needs to call `Studio::System::setListenerAttributes` once per frame for the listener, and to update 3D events with `Studio::EventInstance::set3DAttributes`. It is important to update all orientations before calling `Studio::System::update`. If some orientations are set before `Studio::System::update` and some are set afterwards, then some frames may end up having old positions relative to others. This is particularly important when both the listener and the events are moving fast and together - if there a frame where the listener moves but the event does not it becomes very noticeable.
FMOD Studio 3D Panner

FMOD Studio supports panning events with the FMOD Studio panner on the master track. If there is no panner, the event is considered 2D and the orientation has no effect on the event. It is possible to use other sorts of panners by replacing the FMOD Studio panner with a different type, for example a third party panner.
FMOD Object Panner

The Object panner is a special type of panner that interfaces with Object based output modes such as Dolby Atmos. These output modes accept mono signals with a 3D position and do their own panning and mixing to the final speaker configuration. To use Object panners, the programmer has to specify an output mode that supports Object based panning otherwise the signal will be mixed down at the final stage by FMOD.

The benefit of the Object panner is that it allows the sound designer to leverage Object based technologies. However it does come at a cost, since the signal leaves the mix at the Object panner and does NOT receive DSP effects on the parent buses like normal panners do. The Object panner automatically bases its volume on the combined volumes of parant buses for basic mixing, but no complex effects can be used. For this reason the mix has to be set up very carefully with knowledge of the limitations of the Object panning.

It is possible for the sound designer to use a mixture of normal 3D panned events and Object panned 3D events. Normal events will have signal going through the mixer hierarchy, and Object based events will have signal that leaves the mix at the master track. As far as the programming API goes, both sorts of events are treated exactly the same.
Automatic Parameters

FMOD Studio supports setting automations based on parameters that automatically update based on position. For example, the sound designer could add a volume automation based on Distance, with a 2D panning that is automated on the Direction parameter. The event is still considered 3D in that case, even if it has no panner on the master track.

An event may have both a 3D panner on the master track, as well as an automation based on a Distance parameter. As the event and listener moves, both the panner and the automation will be updated.
Multiple listeners

FMOD Studio supports multiple listeners. Call \texttt{Studio::System::setNumListeners} to set the number of listeners, and use \texttt{Studio::System::setListenerAttributes} to set the orientations for listeners, with an index for the listener.

Studio panning for multiple listeners

Consider the case of an event with three nearby listeners. In this case, listener A is slightly closer to the event than B, and C is the furthest away, outside the max distance of the event.

The Studio 3D panner will take listener A and B into account. The gain will be based off the closest listener distance (in this case, the distance to listener A). Listener B will have an effect on the panning. However, both A and B agree that the event is to the front, so the final pan matrix will be towards the front speakers. Listener C has no effect on the calculation since it is out of range.
Consider this case where listener A and B have moved and now the event is to the right of A and to the left of B. In this case, the gain will be based on the closest listener distance (which is B), but the pan matrix will have signal out of both the left and the right since both listeners have an effect on the mix. If A moved further away then the contribution of A would diminish and the signal would start to come more out of the left speakers. If A moved further enough away, the signal would smoothly interpolate to just B having an influence on the panning.
A new feature of 1.09 is the ability to set listener weights using `Studio::System::setListenerWeight`. This allows listeners to fade in and out of existence, as well as to allow cross-fading of listeners to a new position. In the following picture, we have 4 listeners. Listener C is out of range so it has no influence, and listener D has 0% weighting so it has no influence either. The remaining two listeners have a weight of 40% and 60%. In this example, perhaps the camera is teleporting to a new position and the game is smoothly interpolating to a new orientation.
The gain is a weighted average between A and B, so it is equivalent to having a distance somewhere between the two listeners. The panning of the signal is a mixture of A and B. A is further away and has a lower weight, so the biggest contribution is due to B, meaning the signal sounds mostly in the front speakers. If you imagine panning from A to B, the signal will smoothly interpolate from the back speakers to the front and get louder when the weights scale from A to B.
Listener Mask

Events can have a mask that specifies which listeners are active for that event. By default all listeners apply to all events. By calling `Studio::EventInstance::setListenerMask`, some listeners can be disabled for that event so that they have no influence on the panning. This could be used to group some events and listeners together and have that set only affected by that one listener. When performing the calculation above, any listener not included in the mask is ignored and is as if it does not exist for that event. It is an error to set a combination of mask and weight such that no listener is active for an event.
Doppler

FMOD events support doppler. The sound designer specifies doppler on a per event basis with a scale, so some events may be affected less than others. It is up to the programmer to correctly specify the listener and event velocity. The scale of doppler can be specified at initialization time using `System::set3DSettings`.

For the case of multiple listeners, the doppler is based on the closest listener. If listener has a weight then it is a combination of the closest listeners up to 100%. For example if there were three listeners at increasing distance with weight of 60%, 60% and 60%, then the doppler would be calculated from 60% of the first listener, 40% of the second, and 0% of the third.
Automatic Parameters and multiple listeners

For the case of multiple listeners, the FMOD Studio automatic parameters are based on the closest listener. If listener has a weight then it is a combination of the closest listeners up to 100%. For example if there were three listeners at increasing distance with weight of 60%, 60% and 60%, then the automatic parameters would be calculated from 60% of the first listener, 40% of the second, and 0% of the third.
Interface with Low Level API

When calling `Studio::System::setNumListeners` and `Studio::System::setListenerAttributes`, there is no need to call the equivalent low-level functions `System::set3DNumListeners` and `System::set3DListenerAttributes`. FMOD Studio will pass the information into the low-level API automatically. That means it is possible to have a mixture of FMOD Studio 3D Events and low-level 3D Channels playing at the same time.
Firelight Technologies FMOD Studio API
Studio Banks

Introduction

This section explains the concepts of Studio Banks and how loading works.
Bank Layout

The FMOD Studio Bank file contains event metadata and sound data in the one file.

Loading a bank will load all metadata, which contains information about all the events, parameters, and other data needed for all events associated with that bank.

The sound sample data has two different types, normal sample data, and streaming sound data. Normal sample data can be loaded per event type. Streaming data is streamed in on demand as events are played, and is never fully loaded ahead of time. Streaming data is good for music, voice over, and other sounds which are of a long duration. Whether sound data is streamed or not is set up by the designer in FMOD Studio and cannot be changed at runtime.
**Bank Loading**

Banks are loaded by calling `Studio::System::loadBankFile`. They are unloaded by `Studio::Bank::unload`.

Bank loading can be controlled with the `FMOD_STUDIO_LOAD_BANK_FLAGS`. When loading banks with `FMOD_STUDIO_LOAD_BANK_NORMAL`, the function will not return until the bank has completed loading. When using the `FMOD_STUDIO_LOAD_BANK_NONBLOCKING` flag, the load bank function will return before the bank has completed loading.

As soon as a bank has completed loaded, all the metadata in it can be accessed. This means that event descriptions can be found with `Studio::System::getEvent`, and instances created from those descriptions. The bank loading state can be queried with `Studio::Bank::getLoadingState`. 
Sample Data

Sample data is loaded from one of the three actions:

- `Studio::Bank::loadSampleData`
- `Studio::EventDescription::loadSampleData`
- `Studio::EventDescription::createInstance`

For cases where most or all of the events may play at any time, then loading calling `Studio::Bank::loadSampleData` to load all data up front may be the best approach. Once the bank sample data has loaded, then all event instances can be created or destroyed and use that existing data immediately. However, it does have the highest memory overhead. Repeated calls to `Studio::Bank::loadSampleData` are reference counted, and the bank's sample data is only unloaded when `Studio::Bank::unloadSampleData` has been called an equal number of times.

Sample data can be loaded for selected event using `Studio::EventDescription::loadSampleData`. It is best to load the sample data ahead of time, so that the event's sound sample data is ready when needed. For cases of very common events, the sample data could be loaded for the duration of the game or level. For less common events, the sample data may be loaded in or out as needed. Repeated calls to `Studio::EventDescription::loadSampleData` are reference counted, and the bank's sample data is only unloaded when `Studio::EventDescription::unloadSampleData` has been called an equal number of times, or if the entire bank is unloaded.

If neither of these API calls have been made, then sample data will be loaded if instances exist for that event. The sample loading state can be queried with `Studio::EventDescription::getSampleLoadingState`. If the event's sample data has not yet finished, the event can still be started with `Studio::EventInstance::start`. However, some sounds play slightly later than usual because the sound data may not have completed.

The automatic loading of sample data is the simplest approach and uses the least amount of memory. However it has the following disadvantages:

- Sample data will only start loading when the instance is created, which may
be just before `Studio::EventInstance::start` is called.

- Sample data will only stay loaded for as long as at least one instance exists.

For the case of one-shots, this may mean that the sample data is constantly loaded and unloaded whenever a one-shot plays, which is not a good approach. For these sort of common sounds, it is better to call `Studio::EventDescription::loadSampleData` so the sample data stays in memory rather than constantly unloading then reloading it.

The three approaches to bank loading can be combined. The sample data will stay loaded for as long as at least of the three conditions are met.

**Idle Pool**

For users who don't explicitly load sample data, sounds will be loaded and unloaded on demand. To help avoid unnecessary file access, there is an idle pool for recently used sounds. When a sound is no longer needed (e.g. due to an event instance finishing), its sample data will be placed into the idle pool to be reused later if needed.

By default, the idle pool is set to 256kB in size. This can be customized via the `FMOD_STUDIO_ADVANCEDSETTINGS.idleSampleDataPoolSize` field.
Streaming Data

Streaming data is automatically loaded on demand when needed. There is no API for it since it cannot be preloaded or unloaded.
**Strings Bank**

The Strings bank is a special bank which contains the string lookup of event path to GUID. The strings bank functions identically to a normal bank except that it never contains sample or streaming sound data.
Bank Unload

Banks can be unloaded by calling Studio::Bank::unload. Unloading a bank will free all sample data, invalidate the events descriptions belonging to that bank, and destroy associated instances.
Firelight Technologies FMOD Studio API
Studio Thread Overview

Introduction

This section will describe how Studio execution works in regards to threads.
**Studio Synchronous Mode**

If `Studio::System::initialize` is called with `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE`, then Studio will be created in synchronous mode. In this mode, all Studio API commands are executed during `Studio::System::update`.

As part of that Studio update, it will automatically call the low level `System::update` to ensure that the low level system is updated properly.

The above diagram shows Studio commands being called from the game thread in Studio. It also shows the low level mixer thread, which is triggered based on the hardware output device. The low level mixer thread normally has a period of 5ms, 10ms, or 20ms, depending on the platform. It can also be customised with `System::setDSPBufferSize` and `System::setSoftwareFormat`.

When running in this mode, Studio must deal with the fact that the low level mix can execute at any time. For instance, an event may have two timelocked instruments that should start at the same time. Studio schedules sounds a mix block later so that even if the mix jumps in, all scheduled events will occur in the same mix block.
Studio Asynchronous Mode

The default operation is for Studio to create its own asynchronous thread for execution. In this mode, Studio API commands are enqueued and executed in the Studio asynchronous thread. The commands are batched up so that they are only sent to the asynchronous thread at the end of the next `Studio::System::update`. This prevents some Studio commands from executing earlier than others, which could cause glitches. For instance, if an event position is updated, and the listener position is updated, those two commands will always be executed together.

In asynchronous mode, the Studio processing occurs every 20ms and is triggered off the low level mixer. The low level mix is split into parts, premix, midmix and postmix. It is the low level premix that executes any enqueued low level commands and updates DSP clocks. By triggering the asynchronous Studio processing at the end of the premix, Studio can assume that the mix isn't going to
jump in as the asynchronous update is executing. Unlike the first case, Studio can also assume that the update will be called in a timely manner, even if the game's main thread has a framerate spike.

The size of the Studio asynchronous command buffer can be customised by calling `Studio::System::setAdvancedSettings`. It there is not enough space for commands, then a stall will occur until the asynchronous update has consumed enough commands. `Studio::System::getBufferUsage` can be used to measure if any stalls have occurred due to the command buffer not being large enough.
Game Controlled Worker Thread

Another command situation is for the game to have its own worker thread that invokes Studio using `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE`. This is very similar to the first diagram, except that execution is in a worker rather than the game thread. It is up to the game thread how it wishes to synchronize with the rest of the game. It could be triggered per game frame, or with a fixed period.

In this mode, it is up to the developer to ensure that commands are not split across system updates. For example, consider the case where the game thread issues commands for the worker thread, and the worker thread wakes up periodically to execute those commands. In that case, the worker thread may wake up and execute some commands but not others, causing subtle issues with the sound playback. Instead, the commands to the worker thread should be batched up to avoid slicing commands. Or even better, just use the inbuilt asynchronous mode to do the command batching instead.
Firelight Technologies FMOD Studio API
Examples

The following pages contain information about the programmer examples.
Firelight Technologies FMOD Studio API
Low Level Examples

The following pages contain information about the programmer examples for FMOD Low Level.
Firelight Technologies FMOD Studio API
3D Example

This example shows how to basic 3D positioning of sounds.

Location

The example is located at api/lowlvel/examples/3d.cpp.
Channel Groups Example

This example shows how to put channels into channel groups, so that you can affect a group of channels at a time instead of just one.

Location

The example is located at api/lowlvel/examples/channel_groups.cpp.
Firelight Technologies FMOD Studio API
Convolution Reverb Example

This example shows how to set up a convolution reverb DSP as a global DSP unit that can be routed into by multiple separate channels.

Convolution reverb uses data from a real world locations called an "Impulse Response" to model the reflection of audio waves back to a listener.

Impulse Response is based on "St Andrew's Church" by

www.openairlib.net
Audiolab, University of York
Damian T. Murphy
http://www.openairlib.net/auralizationdb/content/st-andrews-church

licensed under Attribution Share Alike Creative Commons license
http://creativecommons.org/licenses/by-sa/3.0/

Anechoic sample "Operatic Voice" by

www.openairlib.net
http://www.openairlib.net/anechoicdb/content/operatic-voice

licensed under Attribution Share Alike Creative Commons license
http://creativecommons.org/licenses/by-sa/3.0/

Features Demonstrated

- **FMOD_DSP_CONVOLUTION_REVERB**
- **DSP::addInput**

Location

The example is located at api/lowlevel/examples/convolution_reverb.cpp.
Firelight Technologies FMOD Studio API
Custom DSP Example

This example shows how to add a user created DSP callback to process audio data. The read callback is executed at runtime, and can be added anywhere in the DSP network.

Location

The example is located at api/lowlevel/examples/dsp_custom.cpp.
Firelight Technologies FMOD Studio API
DSP Effect Per Speaker Example

This example shows how to manipulate a DSP network and as an example, creates 2 DSP effects, splitting a single sound into 2 audio paths, which it then filters separately.

To only have each audio path come out of one speaker each, \texttt{DSPConnection::setMixMatrix} is used just before the 2 branches merge back together again.

For more speakers:

- Use \texttt{System::setSoftwareFormat}
- Create more effects, currently 2 for stereo (lowpass and highpass), create one per speaker.
- Under the 'Now connect the 2 effects to channeldsp head.' section, connect the extra effects by duplicating the code more times.
- Filter each effect to each speaker by calling \texttt{DSPConnection::setMixMatrix}. Expand the existing code by extending the matrices from 2 in and 2 out, to the number of speakers you require.

Location

The example is located at api/lowllevel/examples/dsp_effect_per_speaker.cpp.
Firelight Technologies FMOD Studio API
Plug-in Inspector Example

This example shows how to enumerate loaded plug-ins and their parameters.

Location

The example is located at api/lowlowlevel/examples/dsp_inspector.cpp.
Firelight Technologies FMOD Studio API
Effects Example

This example shows how to apply some of the built-in software effects to sounds by applying them to the master channel group. All software sounds played here would be filtered in the same way. To filter per channel, and not have other channels affected, simply apply the same functions to the FMOD::Channel instead of the FMOD::ChannelGroup.

Location

The example is located at api/lowlowlevel/examples/effects.cpp.
Firelight Technologies FMOD Studio API
Gapless Playback Example

This example shows how to schedule channel playback into the future with sample accuracy. Use several scheduled channels to synchronize 2 or more sounds.

**Location**

The example is located at api/lowlvel/examples/gapless_playback.cpp.
Firelight Technologies FMOD Studio API
Generate Tone Example

This example shows how to play generated tones using System::playDSP instead of manually connecting and disconnecting DSP units.

Location

The example is located at api/lowlvel/examples/generate_tone.cpp.
Firelight Technologies FMOD Studio API
Granular Synthesis Example

This example shows how you can play a string of sounds together without gaps, using the setDelay command, to produce a granular synthesis style truck engine effect.

The basic operation is:

- Play 2 sounds initially at the same time, the first sound immediately, and the 2nd sound with a delay calculated by the length of the first sound.
- Call setDelay to initiate the delayed playback. setDelay is sample accurate and uses -output- samples as the time frame, not source samples. These samples are a fixed amount per second regardless of the source sound format, for example, 48000 samples per second if FMOD is initialized to 48khz output.
- Output samples are calculated from source samples with a simple source->output sample rate conversion. i.e. sound_length *= output_rate
  sound_length /= sound_frequency
- When the first sound finishes, the second one should have automatically started. This is a good oppurtunity to queue up the next sound. Repeat step 2.
- Make sure the framerate is high enough to queue up a new sound before the other one finishes otherwise you will get gaps.

These sounds are not limited by format, channel count or bit depth like the realtimestitching example is, and can also be modified to allow for overlap, by reducing the delay from the first sound playing to the second by the overlap amount.

#define USE_STREAMS = Use 2 stream instances, created while they pla
#define USE_STREAMS = Use 6 static wavs, all loaded into memory.

Location

The example is located at api/lowlevel/examples/granular_synth.cpp.
Firelight Technologies FMOD Studio API
Load From Memory Example

This example is simply a variant of the Play Sound Example, but it loads the data into memory then uses the 'load from memory' feature of System::createSound.

Location

The example is located at api/lowlve/lowlevel/examples/load_from_memory.cpp.
Firelight Technologies FMOD Studio API
Multiple Speaker Example

This example shows how to play sounds in multiple speakers, and also how to even assign sound subchannels, such as those in a stereo sound to different individual speakers.

Location

The example is located at api/lowlevel/examples/multiple_speaker.cpp.
Firelight Technologies FMOD Studio API
Multiple System Example

This example shows how to play sounds on two different output devices from the same application. It creates two FMOD::System objects, selects a different sound device for each, then allows the user to play one sound on each device.

Note that sounds created on device A cannot be played on device B and vice versa.

Location

The example is located at api/lowlvel/examples/multiple_system.cpp.
Firelight Technologies FMOD Studio API
Net Stream Example

This example shows how to play streaming audio from an Internet source

Location

The example is located at api/lowlevel/examples/net_stream.cpp.
Firelight Technologies FMOD Studio API
Play Sound Example

This example shows how to simply load and play multiple sounds, the simplest usage of FMOD. By default FMOD will decode the entire file into memory when it loads. If the sounds are big and possibly take up a lot of RAM it would be better to use the FMOD_CREATESTREAM flag, this will stream the file in realtime as it plays.

Location

The example is located at api/lowlvel/examples/play_sound.cpp.
Firelight Technologies FMOD Studio API
Play Stream Example

This example shows how to simply play a stream such as an MP3 or WAV. The stream behaviour is achieved by specifying `FMOD_CREATESTREAM` in the call to `System::createSound`. This makes FMOD decode the file in realtime as it plays, instead of loading it all at once which uses far less memory in exchange for a small runtime CPU hit.

Location

The example is located at `api/lowlvel/examples/play_stream.cpp`. 
Firelight Technologies FMOD Studio API
Record example

This example shows how to record continuously and play back the same data while keeping a specified latency between the two. This is achieved by delaying the start of playback until the specified number of milliseconds has been recorded. At runtime the playback speed will be slightly altered to compensate for any drift in either play or record drivers.

Location

The example is located at api/lowlvel/examples/record.cpp.
Firelight Technologies FMOD Studio API
Record enumeration example

This example shows how to enumerate the available recording drivers on this device. It demonstrates how the enumerated list changes as microphones are attached and detached. It also shows that you can record from multi mics at the same time.

Please note to minimize latency care should be taken to control the number of samples between the record position and the play position. Check the record example for details on this process.

Location

The example is located at api/lowlvel/examples/record Enumeration.cpp.
Firelight Technologies FMOD Studio API
**User Created Sound Example**

This example shows how to create a sound with data filled by the user. It shows a user-created static sample, followed by a user-created stream. The former allocates all memory needed for the sound and is played back as a static sample, while the latter streams the data in chunks as it plays, using far less memory.

**Location**

The example is located at api/lowlvel/examples/user_created_sound.cpp.
Firelight Technologies FMOD Studio API
Raw Codec Plugin Example

This example shows how to create a codec that reads raw PCM data.

1. The codec can be compiled as a DLL, using the reserved function name 'FMODGetCodecDescription' as the only export symbol, and at runtime, the dll can be loaded in with System::loadPlugin.

2. Alternatively a codec of this type can be compiled directly into the program that uses it, and you just register the codec into FMOD with System::registerCodec. This puts the codec into the FMOD system, just the same way System::loadPlugin would if it was an external file.

3. The 'open' callback is the first thing called, and FMOD already has a file handle open for it. In the open callback you can use FMOD_CODEC_STATE::fileread / FMOD_CODEC_STATE::fileseek to parse your own file format, and return FMOD_ERR_FORMAT if it is not the format you support. Return FMOD_OK if it succeeds your format test.

4. When an FMOD user calls System::createSound or System::createStream, the 'open' callback is called once after FMOD tries to open it as many other types of file. If you want to override FMOD's internal codecs then use the 'priority' parameter of System::loadPlugin or System::registerCodec.

5. In the open callback, tell FMOD what sort of PCM format the sound will produce with the FMOD_CODEC_STATE::waveformat member.

6. The 'close' callback is called when Sound::release is called by the FMOD user.

7. The 'read' callback is called when System::createSound or System::createStream wants to receive PCM data, in the format that you specified with FMOD_CODEC_STATE::waveformat. Data is interleaved as described in the terminology section of the FMOD API documentation. When a stream is being used, the read callback will be called repeatedly, using a size value determined by the decode buffer size of the stream. See FMOD_CREATESOUNDEXINFO or FMOD_ADVANCEDEDSETTINGS.
8. The 'seek' callback is called when `Channel::setPosition` is called, or when looping a sound when it is a stream.

**Location**

The example is located at api/lowlevel/examples/plugins/fmod_codec_raw.cpp.
Firelight Technologies FMOD Studio API
Distance Filter DSP Plugin Example

This example shows how to create a distance filter DSP effect.

Location

The example is located at api/lowlevel/examples/plugins/fmod_distance_filter.cpp.
Firelight Technologies FMOD Studio API
Gain DSP Plugin Example

This example shows how to create a simple gain DSP effect.

Location

The example is located at api/lowlvel/examples/plugins/fmod_gain.cpp.
Firelight Technologies FMOD Studio API
Plugin Example

This example shows how to create a plugin effect.

Location

The example is located at api/lowlevel/examples/plugins/fmod_noise.cpp.
Firelight Technologies FMOD Studio API
Studio Examples

The following pages contain information about the programmer examples for FMOD Studio.
Firelight Technologies FMOD Studio API
Event 3D Example

This example demonstrates how to position events in 3D for spatialization.

Location

The example is located at api/studio/examples/3d.cpp.
Firelight Technologies FMOD Studio API
Event 3D Multi-Listener Example

This example demonstrates how use listener weighting to crossfade listeners in and out.

Location

The example is located at api/studio/examples/3d_multi.cpp.
Firelight Technologies FMOD Studio API
Event Parameter Example

This example demonstrates how to control event playback using game parameters.

Location

The example is located at api/studio/examples/event_parameter.cpp.
Firelight Technologies FMOD Studio API
Load Banks Example

This example demonstrates loading banks via file, memory, and user callbacks.

The banks that are loaded are:

- Character.bank (file)
- UI_Menu.bank (memory)
- Vehicles.bank (memory-point)
- Weapons.bank (custom)

The loading and unloading is asynchronous, and we displays the current state of each bank as loading is occurring.

See Also

- Studio::System::loadBankFile
- Studio::System::loadBankMemory
- Studio::System::loadBankCustom
- Studio::Bank::loadSampleData
- Studio::Bank::getLoadingState
- Studio::Bank::getSampleLoadingState
- Studio::Bank::getUserData
- Studio::Bank::setUserData

Location

The example is located at api/studio/examples/load_banks.cpp.
Firelight Technologies FMOD Studio API
Music Callback Example

This example demonstrates beat and named marker callbacks when playing music.

See Also

- [Studio::EventInstance::setCallback](#)
- [FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_MARKER](#)
- [FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_BEAT](#)

Location

The example is located at api/studio/examples/music_callbacks.cpp.
Firelight Technologies FMOD Studio API
Object Panning Example

This example demonstrates the FMOD object panner. The usage is completely transparent to the API, the only difference is how the event is authored in the FMOD Studio tool.

To hear the difference between object panning and normal panning this example has two events (one configured with the normal panner, and one with the object panner). As they move around the listener you may toggle between panning method and two different sounds.

Object panning requires compatible hardware such as a Dolby Atmos amplifier or a Playstation VR headset. For cases when the necessary hardware is not available FMOD will fallback to standard 3D panning.

NOTE! Currently the Atmos output mode requires a dll to be supplied via support. Atmos access needs to be granted via Dolby first. Write to support@fmod.com for more.

Location

The example is located at api/studio/examples/objectpan.cpp.
Firelight Technologies FMOD Studio API
Programmer Sound Example

This example demonstrates how to implement the programmer sound callback to play an event that has a programmer specified sound.

See Also

Studio::EventInstance::setCallback

Location

The example is located at api/studio/examples/programmer_sound.cpp.
Firelight Technologies FMOD Studio API
API Recording Example

This example shows recording and playback functionality, allowing the user to trigger some sounds and then play back what they have recorded. The provided functionality is intended to assist in debugging.

Location

The example is located at api/studio/examples/recording_playback.cpp.
Firelight Technologies FMOD Studio API
Simple Event Example

This example demonstrates the various ways of playing an event.

**Explosion Event**

This event is played as a one-shot and released immediately after it has been created.

**Looping Ambience Event**

A single instance is started or stopped based on user input.

**Cancel Event**

This instance is started and if already playing, restarted.

**Location**

The example is located at api/studio/examples/simple_event.cpp.
Firelight Technologies FMOD Studio API
Reference

The following pages contain the complete reference API documentation.
Firelight Technologies FMOD Studio API
Low Level API

Classes  Functions
Callbacks
Structures
Defines
Enumerations
Firelight Technologies FMOD Studio API
Classes

System  Sound
ChannelControl
Channel
ChannelGroup
SoundGroup
DSP
DSPConnection
Geometry
Reverb3D
Firelight Technologies FMOD Studio API
System

The main object for the FMOD Low Level System.
Functions

System::attachChannelGroupToPort System::attachFileSystem
System::close
System::createChannelGroup
System::createDSP
System::createDSPByPlugin
System::createDSPByType
System::createGeometry
System::createReverb3D
System::createSound
System::createSoundGroup
System::createStream
System::detachChannelGroupFromPort
System::get3DListenerAttributes
System::get3DNumListeners
System::get3DSettings
System::getAdvancedSettings
System::getCPUUsage
System::getChannel
System::getChannelsPlaying
System::getDSPBufferSize
System::getDSPInfoByPlugin
System::getDefaultMixMatrix
System::getDriver
System::getDriverInfo
System::getFileUsage
System::getGeometryOcclusion
System::getGeometrySettings
System::getMasterChannelGroup
System::getMasterSoundGroup
System::getNestedPlugin
System::getNetworkProxy
System::getNetworkTimeout
System::getNumDrivers
System::getNumNestedPlugins
System::getNumPlugins
System::getOutput
System::getOutputByPlugin
System::getOutputHandle
System::getPluginHandle
System::getPluginInfo
System::getRecordDriverInfo
System::getRecordNumDrivers
System::getRecordPosition
System::getReverbProperties
System::getSoftwareChannels
System::getSoftwareFormat
System::getSoundRAM
System::getSpeakerModeChannels
System::getSpeakerPosition
System::getStreamBufferSize
System::getUserData
System::getVersion
System::init
System::isRecording
System::loadGeometry
System::loadPlugin
System::lockDSP
System::mixerResume
System::mixerSuspend
System::playDSP
System::playSound
System::recordStart
System::recordStop
System::registerCodec
System::registerDSP
System::registerOutput
System::release
System::set3DListenerAttributes
System::set3DNumListeners
System::set3DRolloffCallback
System::set3DSettings
System::setAdvancedSettings
System::setCallback
System::setDSPBufferSize
System::setDriver
System::setFileSystem
System::setGeometrySettings
System::setNetworkProxy
System::setNetworkTimeout
System::setOutput
System::setOutputByPlugin
System::setPluginPath
System::setReverbProperties
System::setSoftwareChannels
System::setSoftwareFormat
System::setSpeakerPosition
System::setStreamBufferSize
System::setUserData
System::unloadPlugin
System::unlockDSP
System::update
Remarks

When using FMOD Studio, this system object will be automatically instantiated as part of `Studio::System::initialize`. 
See Also

- [Studio::System::getLowLevelSystem](#)
Firelight Technologies FMOD Studio API
System::attachChannelGroupToPort

Route the signal from a channel group into a separate audio port on the output driver.

C++ Syntax

FMOD_RESULT System::attachChannelGroupToPort(
    FMOD_PORT_TYPE portType,
    FMOD_PORT_INDEX portIndex,
    FMOD::ChannelGroup *channelgroup,
    bool passThru
);

C Syntax

FMOD_RESULT FMOD_System_AttachChannelGroupToPort(
    FMOD_SYSTEM *system,
    FMOD_PORT_TYPE portType,
    FMOD_PORT_INDEX portIndex,
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL passThru
);

C# Syntax

RESULT System.attachChannelGroupToPort(
    uint portType,
    ulong portIndex,
    ChannelGroup channelgroup,
    bool passThru = false
);

JavaScript Syntax

System.attachChannelGroupToPort(  
    portType,  
    portIndex,  
    channelgroup,  
    passThru  
);
Parameters

portType
Output driver specific audio port type. See extra platform specific header (if it exists) for port numbers, i.e. fmod_psvita.h, fmod_wiiu.h, fmodorbis.h

portIndex
Output driver specific index of the audio port

channelgroup
Channel group to route away to the new port

passThru
If true the signal will continue to be passed through to the main mix, if false the signal will be entirely to the designated port.
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

Note that an FMOD port is a hardware specific reference, to hardware devices that exist on only certain platforms (like a console headset, or dedicated hardware music channel for example). It is not supported on all platforms.
See Also

- System::detachChannelGroupFromPort
Firelight Technologies FMOD Studio API
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).

C++ Syntax

```cpp
FMOD_RESULT System::attachFileSystem(
    FMOD_FILE_OPEN_CALLBACK useropen,
    FMOD_FILE_CLOSE_CALLBACK userclose,
    FMOD_FILE_READ_CALLBACK userread,
    FMOD_FILE_SEEK_CALLBACK userseek
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_AttachFileSystem(
    FMOD_SYSTEM *system,
    FMOD_FILE_OPEN_CALLBACK useropen,
    FMOD_FILE_CLOSE_CALLBACK userclose,
    FMOD_FILE_READ_CALLBACK userread,
    FMOD_FILE_SEEK_CALLBACK userseek
);
```

C# Syntax

```csharp
RESULT System.attachFileSystem(
    FILE_OPENCALLBACK useropen,
    FILE_CLOSECALLBACK userclose,
    FILE_READCALLBACK userread,
    FILESEEKCALLBACK userseek
);
```

JavaScript Syntax

```javascript
System.attachFileSystem(
    useropen,
    userclose,
    userread,
```
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 \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

**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_OPEN_CALLBACK
- FMOD_FILE_CLOSE_CALLBACK
- FMOD_FILE_READ_CALLBACK
- FMOD_FILE_SEEK_CALLBACK
Firelight Technologies FMOD Studio API
**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);
```

**C# Syntax**

```csharp
RESULT System.close();
```

**JavaScript Syntax**

```javascript
System.close();
```
Return Values

If the function succeeds then the return 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 Studio API
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
);
```

C# Syntax

```csharp
RESULT System.createChannelGroup(
    string name,
    out ChannelGroup channelgroup
);
```

JavaScript Syntax

```javascript
System.createChannelGroup(
    name,
    channelgroup // writes value to channelgroup.v
);
```
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. By default newly created channel groups are parented to the system master channel group. Channel groups can be re-parented using `ChannelGroup::addGroup` to create a hierarchical mixing layout.
See Also

- System::getMasterChannelGroup
- Channel::setChannelGroup
- ChannelGroup::release
- ChannelGroup::addGroup
Firelight Technologies FMOD Studio API
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(
    const FMOD_DSP_DESCRIPTION *description,
    FMOD::DSP **dsp
);
```

C Syntax

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

C# Syntax

```csharp
RESULT System.createDSP(
    ref DSP_DESCRIPTION description,
    out DSP dsp
);
```

JavaScript Syntax

```javascript
System.createDSP(
    description,
    dsp // writes value to dsp.val
);
```
Parameters

description
Address of an `FMOD_DSP_DESCRIPTION` structure containing information about the unit to be created. Some members of `FMOD_DSP_DESCRIPTION` are referenced directly inside FMOD so the structure should be allocated statically or at least remain in memory for the lifetime of the system.

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 `ChannelGroup::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
- ChannelGroup::addDSP
- Channel::addDSP
- DSP::addInput
- DSP::setActive
Firelight Technologies FMOD Studio API
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

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

C Syntax

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

C# Syntax

RESULT System.createDSPByPlugin(
  uint handle,
  out DSP dsp
);

JavaScript Syntax

System.createDSPByPlugin(
  handle,
  dsp // writes value to dsp.val
);
Parameters

handle
  Handle to a pre-existing DSP plugin, loaded by System::loadPlugin.

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 `ChannelGroup::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
- ChannelGroup::addDSP
- Channel::addDSP
- DSP::addInput
- DSP::setActive
Firelight Technologies FMOD Studio API
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

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

C Syntax

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

C# Syntax

RESULT System.createDSPByType(
   DSP_TYPE type,
   out DSP dsp
);

JavaScript Syntax

System.createDSPByType(
   type,
   dsp // writes value to dsp.val
);
**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 `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::loadPlugin
- Channel::addDSP
- ChannelGroup::addDSP
- DSP::addInput
- DSP::setActive
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.createGeometry(
   int maxpolygons,
   int maxvertices,
   out Geometry geometry
);
```

JavaScript Syntax

```javascript
System.createGeometry(
   maxpolygons,
   maxvertices,
   geometry // writes value to geometry.val
);
```
Parameters

maxpolygons
  Maximum number of polygons within this object.
maxvertices
  Maximum number of vertices within this object.
gometry
  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 Studio API
System::createReverb3D

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. 1 Physical reverb object is used for all 3D reverb objects (slot 0 by default).

C++ Syntax

FMOD_RESULT System::createReverb3D(
    FMOD::Reverb3D **reverb
);

C Syntax

FMOD_RESULT FMOD_System_CreateReverb3D(
    FMOD_SYSTEM *system,
    FMOD_REVERB3D **reverb
);

C# Syntax

RESULT System.createReverb3D(
    out Reverb3D reverb
);

JavaScript Syntax

System.createReverb3D(
    reverb
    // writes value to reverb.val
);
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.
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, no reverb is applied.

In FMOD Ex a special 'ambient' reverb setting was used when outside the influence of all reverb spheres. This function no longer exists.
In FMOD Studio System::setReverbProperties can be used to create an alternative reverb that can be used for 2D and background global reverb.
To avoid this reverb intefering with the reverb slot used by the 3D reverb, 2D reverb should use a different slot id with System::setReverbProperties, otherwise FMOD_ADVANCEDSETTINGS::reverb3Dinstance can also be used to place 3D reverb on a different physical reverb slot. Use Channel::setReverbProperties or ChannelGroup::setReverbProperties to turn off reverb for 2D sounds (ie set wet = 0).

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.

Note about physical reverb DSP unit allocation. To remove the DSP unit and the associated CPU cost, first make sure all 3D reverb objects are released. Then call System::setReverbProperties with the 3D reverb's slot ID (default is 0) with a property point of 0 or NULL, to signal that the physical reverb instance should be deleted.
If a 3D reverb is still present, and System::setReverbProperties function is called to free the physical reverb, the 3D reverb system will immediately recreate it upon the next System::update call.

Note that the 3D reverb system will not affect Studio events unless it is explicitly enabled by calling Studio::EventInstance::setReverbLevel on each event instance.
See Also

- Reverb3D::release
- System::setReverbProperties
- System::getReverbProperties
- FMOD_ADVANCEDSETTINGS
- System::update

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
System::createSound

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

C++ Syntax

FMOD_RESULT System::createSound(
    const char *name_or_data,
    FMOD_MODE mode,
    FMOD_CREATESONDEXINFO *exinfo,
    FMOD::Sound **sound
);

C Syntax

FMOD_RESULT FMOD_System_CreateSound(
    FMOD_SYSTEM *system,
    const char *name_or_data,
    FMOD_MODE mode,
    FMOD_CREATESONDEXINFO *exinfo,
    FMOD_SOUND **sound
);

C# Syntax

RESULT System.createSound(
    string name,
    MODE mode,
    out Sound sound
);

JavaScript Syntax

System.createSound(
    name_or_data,
    mode,
    exinfo,
    sound
    // writes value to sound.val
);
**Parameters**

name_or_data
Name of the file or URL to open encoded in a UTF-8 string, 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_CREATESOUNDEXINFO** 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_CREATETEMPLATE**) FMOD will try to load and decompress the whole sound into memory! Use **FMOD_CREATETEMPLATE** to open it as a stream and have it play back in realtime! **FMOD_CREATETEMPLATE** 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_CREATETEMPLATE** 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_CREATETEMPLATE**. This is supported with MPEG (mp2/mp3), ADPCM/FADPCM, XMA, AT9 and FSB Vorbis files only. This is useful for those who want realtime compressed sound effects, but not the overhead of disk access.
- 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.

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.

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.

**JavaScript only :**

If FMOD.NON_BLOCKING is passed into mode, FMOD.ERR_UNSUPPORTED will be returned
See Also

- **FMOD_MODE**
- **FMOD_CREATESOUNDEXINFO**
- **Sound::getOpenState**
- **System::setStreamBufferSize**
- **Channel::setPan**
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.createSoundGroup(
    string name,
    out SoundGroup soundgroup
);
```

JavaScript Syntax

```javascript
System.createSoundGroup(
    name,
    soundgroup // writes value to soundgroup.val
);
```
Parameters

name
   Name of sound group.
soundgroup
   Address of a variable to receive a pointer to a sound group.
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

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 Studio API
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

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

C Syntax

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

C# Syntax

RESULT System.createStream(
    string name,
    MODE mode,
    out Sound sound
);

JavaScript Syntax

System.createStream(
    name_or_data,
    mode,
    exinfo,
    sound // writes value to sound.val
);
Parameters

name_or_data
   Name of the file or URL to open encoded in a UTF-8 string.

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

exinfo
   Pointer to a FMOD_CREATESOUNDEXINFO 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/FADPCM, XMA, AT9 and FSB Vorbis files only. This is useful for those who want realtime compressed sound effects, but not the overhead of disk access.
- 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.

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 Studio API
**System::detachChannelGroupFromPort**

Disconnect a channel group from a and route audio back to the default port of the output driver

**C++ Syntax**

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

**C Syntax**

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

**C# Syntax**

```csharp
RESULT System.detachChannelGroupFromPort(
    ChannelGroup channelgroup
);
```

**JavaScript Syntax**

```javascript
System.detachChannelGroupFromPort(
    channelgroup
);
```
Parameters

channelgroup
  Channel group to route away back to the default audio port
Return Values

If the function succeeds then the return 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::attachChannelGroupToPort](#)
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.get3DListenerAttributes(
    int listener,
    out VECTOR pos,
    out VECTOR vel,
    out VECTOR forward,
    out VECTOR up
);
```

JavaScript Syntax

```javascript
System.get3DListenerAttributes(
    listener,
    pos, // writes value to pos.val
```
vel,  // writes value to vel.val
forward,  // writes value to forward.val
up  // writes value to up.val
);
**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.
Remarks

NOTE! Users of the Studio API should call Studio::System::getListenerAttributes instead of this function.
See Also

- System::set3DListenerAttributes
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT System.get3DNumListeners(
    out int numlisteners
);
```

**JavaScript Syntax**

```javascript
System.get3DNumListeners(
    numlisteners // writes value to numlisteners.v
);
```
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.
Remarks

**NOTE!** Users of the Studio API should call `Studio::System::getNumListeners` instead of this function.
See Also

- System::set3DNumListeners
Firelight Technologies FMOD Studio API
**System::get3DSettings**

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

**C++ Syntax**

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

**C Syntax**

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

**C# Syntax**

```csharp
RESULT System.get3DSettings(
    out float dopplerscale,
    out float distancefactor,
    out float rolloffscale
);
```

**JavaScript Syntax**

```javascript
System.get3DSettings(
    dopplerscale,             // writes value to dopplerscale.v
distancefactor,            // writes value to distancefactor.v
rolloffscale               // writes value to rolloffscale.v
);
```
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
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
);
```

C# Syntax

```csharp
RESULT System.getAdvancedSettings(
    ref ADVANCEDSETTINGS settings
);
```

JavaScript Syntax

```javascript
System.getAdvancedSettings(
    settings // writes value to settings.val
);
```
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 Studio API
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

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

C Syntax

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

C# Syntax

RESULT System.getCPUUsage(
    out float dsp,
    out float stream,
    out float geometry,
    out float update,
    out float total
);

JavaScript Syntax

System.getCPUUsage(  
    usage // writes value to usage.val
);
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
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT System.getChannel(
    int channelid,
    out Channel channel
);
```

**JavaScript Syntax**

```javascript
System.getChannel(
    channelid,
    channel // writes value to channel.val
);
```
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 Studio API
System::getChannelsPlaying

Retrieves the number of currently playing channels.

C++ Syntax

FMOD_RESULT System::getChannelsPlaying(
    int *channels,
    int *realchannels
);

C Syntax

FMOD_RESULT FMOD_System_GetChannelsPlaying(
    FMOD_SYSTEM *system,
    int *channels,
    int *realchannels
);

C# Syntax

RESULT System.getChannelsPlaying(
    out int channels,
    out int realchannels
);

JavaScript Syntax

System.getChannelsPlaying(
    channels, // writes value to channels.val
    realchannels // writes value to realchannels.v
);
Parameters

channels
Address of a variable that receives the number of playing channels (both real and virtual). Specify 0 or NULL to ignore.

realchannels
Address of a variable that receives the number of playing non-virtual channels. 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::isPlaying
- Channel::isVirtual
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.getDSPBufferSize(
    out uint bufferlength,
    out int numbuffers
);
```

JavaScript Syntax

```javascript
System.getDSPBufferSize(
    bufferlength, // writes value to bufferlength.v
    numbuffers // writes value to numbuffers.val
);
```
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 Studio API
**System::getDSPInfoByPlugin**

Retrieve the description structure for a pre-existing DSP plugin.

**C++ Syntax**

```cpp
FMOD_RESULT System::getDSPInfoByPlugin(
    unsigned int handle,
    const FMOD_DSP_DESCRIPTION **description
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetDSPInfoByPlugin(
    FMOD_SYSTEM *system,
    unsigned int handle,
    const FMOD_DSP_DESCRIPTION **description
);
```

**C# Syntax**

```csharp
RESULT System.getDSPInfoByPlugin(
    uint handle,
    out IntPtr description
);
```

**JavaScript Syntax**

```javascript
System.getDSPInfoByPlugin(
    handle,
    description, // writes value to description.va
);```
Parameters

handle
   Handle to a pre-existing DSP plugin, loaded by System::loadPlugin.
description
   Address of a variable to receive the description structure for the DSP.
Return Values

If the function succeeds then the return 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 Studio API
System::getDefaultMixMatrix

Gets the default matrix used to convert from one speaker mode to another.

**C++ Syntax**

```cpp
FMOD_RESULT System::getDefaultMixMatrix(
    FMOD_SPEAKERMODE sourcespeakermode,
    FMOD_SPEAKERMODE targetspeakermode,
    float *matrix,
    int matrixhop
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetDefaultMixMatrix(
    FMOD_SYSTEM *system,
    FMOD_SPEAKERMODE sourcespeakermode,
    FMOD_SPEAKERMODE targetspeakermode,
    float *matrix,
    int matrixhop
);
```

**C# Syntax**

```csharp
RESULT System.getDefaultMixMatrix(
    SPEAKERMODE sourcespeakermode,
    SPEAKERMODE targetspeakermode,
    float[] matrix,
    int matrixhop
);
```

**JavaScript Syntax**

```javascript
System.getDefaultMixMatrix(
    sourcespeakermode,
    targetspeakermode,
    matrix, // writes value to matrix.val
    matrixhop
);
```
Parameters

sourcespeakermode
The speaker mode being converted from.

targetspeakermode
The speaker mode being converted to.

matrix
The output matrix. Its minimum size in floats must be the number of source channels multiplied by the number of target channels. Source and target channels cannot exceed `FMOD_MAX_CHANNEL_WIDTH`.

matrixhop
The number of source channels in the matrix. Optional. If this is 0, the number of source channels will be derived from 'sourcespeakermode'. Maximum of `FMOD_MAX_CHANNEL_WIDTH`.

Return Values

If the function succeeds then the return value 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 gain for source channel 's' to target channel 't' is matrix[t * matrixhop + s].

If 'sourcespeakermode' or 'targetspeakermode' is FMOD_SPEAKERMODE_RAW, this function will return FMOD_ERR_INVALID_PARAM.
See Also

- System::getSpeakerModeChannels
- FMOD_MAX_CHANNEL_WIDTH
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.getDriver(
    out int driver
);
```

JavaScript Syntax

```javascript
System.getDriver(
    driver // writes value to driver.val
);
```
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 Studio API
**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,
    int *systemrate,
    FMOD_SPEAKERMODE *speakermode,
    int *speakermodechannels
);
```

**C Syntax**

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

**C# Syntax**

```csharp
RESULT System.getDriverInfo(
    int id,
    StringBuilder name,
    int namelen,
    out Guid guid,
    out int systemrate,
    out SPEAKERMODE speakermode,
    out int speakermodechannels
);
```
JavaScript Syntax

System.getDriverInfo(
    id,
    name,
    guid,
    systemrate,
    speakermode,
    speakermodechannels
);
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 encoded in a UTF-8 string. 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.

systemrate
   Address of a variable that receives the sample rate this device operates at. Optional. Specify 0 or NULL to ignore.

speakermode
   Address of a variable that receives the speaker setup this device is currently using. Optional. Specify 0 or NULL to ignore.

speakermodechannels
   Address of a variable that receives the number of channels in the current speaker setup. 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

JavaScript only:

Note: For the "name" parameter, the maximum string length is 512.
See Also

- System::getNumDrivers
- System::setOutput
Firelight Technologies FMOD Studio API
System::getFileUsage

Retrieves information about file reads by FMOD.

C++ Syntax

```cpp
FMOD_RESULT System::getFileUsage(
    long long *sampleBytesRead,
    long long *streamBytesRead,
    long long *otherBytesRead
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetFileUsage(
    FMOD_SYSTEM *system,
    long long *sampleBytesRead,
    long long *streamBytesRead,
    long long *otherBytesRead
);
```

C# Syntax

```csharp
RESULT System.getFileUsage(
    out Int64 sampleBytesRead,
    out Int64 streamBytesRead,
    out Int64 otherBytesRead
);
```

JavaScript Syntax

```javascript
System.getFileUsage(
    sampleBytesRead,   // writes value to sampleBytesRead
    streamBytesRead,   // writes value to streamBytesRead
    otherBytesRead     // writes value to otherBytesRead
);
```
Parameters

sampleBytesRead
Address of a variable that receives the total bytes read from file for loading sample data. Optional. Specify 0 or NULL to ignore.

streamBytesRead
Address of a variable that receives the total bytes read from file for streaming sounds. Optional. Specify 0 or NULL to ignore.

otherBytesRead
Address of a variable that receives the total bytes read for non-audio data such as FMOD Studio banks. 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 values returned are running totals that never reset.
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.getGeometryOcclusion(
    ref VECTOR listener,
    ref VECTOR source,
    out float direct,
    out float reverb
);
```

JavaScript Syntax

```javascript
System.getGeometryOcclusion(
    listener,       // writes value to listener.val
    source,         // writes value to source.val
    direct,         // writes value to direct.val
    reverb          // writes value to reverb.val
);
```
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 Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT System.getGeometrySettings(
    out float maxworldsize
);
```

**JavaScript Syntax**

```javascript
System.getGeometrySettings(
    maxworldsize // writes value to maxworldsize.v
);
```
**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 Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT System.getMasterChannelGroup(
    out ChannelGroup channelgroup
);
```

**JavaScript Syntax**

```javascript
System.getMasterChannelGroup(
    channelgroup // writes value to channelgroup.v
);
```
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 Studio API
System::getMasterSoundGroup

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

C++ Syntax

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

C Syntax

```
FMOD_RESULT FMOD_System_GetMasterSoundGroup(
    FMOD_SYSTEM *system,
    FMOD_SOUNDGROUP **soundgroup
);
```

C# Syntax

```
RESULT System.getMasterSoundGroup(
    out SoundGroup soundgroup
);
```

JavaScript Syntax

```
System.getMasterSoundGroup(
    soundgroup // writes value to soundgroup.val
);
```
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

- `SoundGroup::release`
- `SoundGroup::getSystemObject`
- `SoundGroup::setMaxAudible`
- `SoundGroup::getMaxAudible`
- `SoundGroup::getName`
- `SoundGroup::getNumSounds`
- `SoundGroup::getSound`
- `SoundGroup::getNumPlaying`
- `SoundGroup::setUserData`
- `SoundGroup::getUserData`
System::getNestedPlugin

Returns nested plugin definition for the given index.

**C++ Syntax**

```cpp
FMOD_RESULT System::getNestedPlugin(
    unsigned int handle,
    int index,
    unsigned int *nestedhandle
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetNestedPlugin(
    FMOD_SYSTEM *system,
    unsigned int handle,
    int index,
    unsigned int *nestedhandle
);
```

**C# Syntax**

```csharp
RESULT System.getNestedPlugin(
    uint handle,
    int index,
    out uint nestedhandle
);
```

**JavaScript Syntax**

```javascript
System.getNestedPlugin(
    handle,
    index,
    nestedhandle
); // writes value to nestedhandle.v
```
**Parameters**

handle
   Handle obtained from `System::loadPlugin`.

index
   Index into the list of plugin definitions.

nestedhandle
Return Values

If the function succeeds then the return value 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 used to iterate handles for plugins that have a list of definitions.

For plugins consisting of a single definition, only index 0 is valid and the returned handle is the same as the handle passed in.

See the DSP Plugin API Programmer Topic for more information.
See Also

- System::loadPlugin
- System::getNumNestedPlugins
- FMOD_PLUGINLIST

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
System::getNetworkProxy

Retrieves the URL of the proxy server used in internet streaming.

**C++ Syntax**

```cpp
FMOD_RESULT System::getNetworkProxy(
    char *proxy,
    int proxyleng
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetNetworkProxy(
    FMOD_SYSTEM *system,
    char *proxy,
    int proxyleng
);
```

**C# Syntax**

```csharp
RESULT System.getNetworkProxy(
    StringBuilder proxy,
    int proxyleng
);
```

**JavaScript Syntax**

```javascript
System.getNetworkProxy(
    proxy // writes value to proxy.val
);
```
**Parameters**

**proxy**
Address of a variable that receives the proxy server URL encoded in a UTF-8 string.

**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.
Remarks

JavaScript only:

Note: For the "proxy" parameter, the maximum string length is 512.
See Also

- System::setNetworkProxy
Firelight Technologies FMOD Studio API
System::getNetworkTimeout

Retrieve the timeout value for network streams

C++ Syntax

FMOD_RESULT System::getNetworkTimeout(
    int *timeout
);

C Syntax

FMOD_RESULT FMOD_System_GetNetworkTimeout(
    FMOD_SYSTEM *system,
    int *timeout
);

C# Syntax

RESULT System.getNetworkTimeout(
    out int timeout
);

JavaScript Syntax

System.getNetworkTimeout(
    timeout
    // writes value to timeout.val
);
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 Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT System.getNumDrivers(
    out int numdrivers
);
```

**JavaScript Syntax**

```javascript
System.getNumDrivers(
    numdrivers // writes value to numdrivers.val
);
```
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 Studio API
System::getNumNestedPlugins

Returns the number of plugins nested in the one plugin file.

C++ Syntax

FMOD_RESULT System::getNumNestedPlugins(
    unsigned int handle,
    int *count
);

C Syntax

FMOD_RESULT FMOD_System_GetNumNestedPlugins(
    FMOD_SYSTEM *system,
    unsigned int handle,
    int *count
);

C# Syntax

RESULT System.getNumNestedPlugins(
    uint handle,
    out int count
);

JavaScript Syntax

System.getNumNestedPlugins(
    handle,
    count // writes value to count.val
);
Parameters

handle
  Handle obtained from System::loadPlugin.

count
  Returned number of 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.
Remarks

Plugins normally have a single definition in them, in which case the count is always 1.

For plugins that have a list of definitions, this function returns the number of plugins that have been defined. `System::getNestedPlugin` can be used to find each handle.

See the DSP Plugin API Programmer Topic for more information.
See Also

- System::loadPlugin
- System::getNestedPlugin
- FMOD_PLUGINLIST
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT System.getNumPlugins(
    PLUGINTYPE plugintype,
    out int numplugins
);
```

**JavaScript Syntax**

```javascript
System.getNumPlugins(
    plugintype,
    numplugins
); // writes value to numplugins.val
```
Parameters

plugintype
  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 Studio API
System::getOutput

Retrieves the current output system FMOD is using to address the hardware.

C++ Syntax

FMOD_RESULT System::getOutput(
    FMOD_OUTPUTTYPE *output
);

C Syntax

FMOD_RESULT FMOD_System_GetOutput(
    FMOD_SYSTEM *system,
    FMOD_OUTPUTTYPE *output
);

C# Syntax

RESULT System.getOutput(
    out OUTPUTTYPE output
);

JavaScript Syntax

System.getOutput(
    output
);                 // writes value to output.val
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

- `System::setOutput`
- `FMOD_OUTPUTTYPE`
Firelight Technologies FMOD Studio API
System::getOutputByPlugin

Returns the currently selected output as an id in the list of output plugins.

C++ Syntax

FMOD_RESULT System::getOutputByPlugin(
    unsigned int *handle
);

C Syntax

FMOD_RESULT FMOD_System_GetOutputByPlugin(
    FMOD_SYSTEM *system,
    unsigned int *handle
);

C# Syntax

RESULT System.getOutputByPlugin(
    out uint handle
);

JavaScript Syntax

System.getOutputByPlugin(
    handle
); // writes value to handle.val
Parameters

dhandle
  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 Studio API
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

FMOD_RESULT System::getOutputHandle(
   void **handle
);

C Syntax

FMOD_RESULT FMOD_System_GetOutputHandle(
   FMOD_SYSTEM *system,
   void **handle
);

C# Syntax

RESULT System.getOutputHandle(
   out IntPtr handle
);

JavaScript Syntax

System.getOutputHandle(
   handle
);
Parameters

handle
   Address of a variable that receives the handle to the output mode's native hardware API object (see remarks for supported outputs).
Return Values

If the function succeeds then the return value 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_WAVWRITER Pointer to stdin FILE is returned
FMOD_OUTPUTTYPE_WAVWRITER_NRT Pointer to stdin FILE is returned

FMOD_OUTPUTTYPE_DSOUD Pointer to type IDirectSound8 is returned.
FMOD_OUTPUTTYPE_WINMM Pointer to type HWAVEOUT is returned.
FMOD_OUTPUTTYPE_WASAPI Pointer to type IAudioClient is returned.
FMOD_OUTPUTTYPE_ALSA Pointer to type snd_pcm_t is returned.
FMOD_OUTPUTTYPE_COREAUDIO Handle of type AudioUnit is returned.
FMOD_OUTPUTTYPE_XAUDIO (Xbox360) Pointer to type IXAudio2 is returned. FMOD_OUTPUTTYPE_AUDIOOUT (PS4) Pointer to type int is returned. Handle returned from sceAudioOutOpen.
See Also

- `FMOD_OUTPUTTYPE`
- `System::setOutput`
- `System::init`
Firelight Technologies FMOD Studio API
System::getPluginHandle

Retrieves the handle of a plugin based on its type and relative index. Use System::getNumPlugins to enumerate plugins.

C++ Syntax

FMOD_RESULT System::getPluginHandle(
    FMOD_PLUGINTYPE plugintype,
    int index,
    unsigned int *handle
);

C Syntax

FMOD_RESULT FMOD_System_GetPluginHandle(
    FMOD_SYSTEM *system,
    FMOD_PLUGINTYPE plugintype,
    int index,
    unsigned int *handle
);

C# Syntax

RESULT System.getPluginHandle(
    PLUGINTYPE plugintype,
    int index,
    out uint handle
);

JavaScript Syntax

System.getPluginHandle(
    plugintype,
    index,
    handle // writes value to handle.val
);
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

- **FMOD_PLUGINTYPE**
- **System::getNumPlugins**
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.getPluginInfo(
    uint handle,
    out PLUGINTYPE plugintype,
    StringBuilder name,
    int namelen,
    out uint version
);
```

JavaScript Syntax

```javascript
System.getPluginInfo(
    handle,                   // writes value to plugintype.val
    plugintype,               // writes value to name.val
    name,                     // writes value to version.val
    version
);```
);
Parameters

handle
   Handle to a pre-existing plugin.
plugintype
   Address of a variable that receives the type of the plugin,
   \texttt{FMOD\_PLUGINTYPE\_OUTPUT}, \texttt{FMOD\_PLUGINTYPE\_CODEC} or
   \texttt{FMOD\_PLUGINTYPE\_DSP}.
name
   Address of a variable that receives the name of the plugin.
namelen
   Length in bytes of the target buffer to receive the string.
version
   Address of a variable that receives the version number set by the plugin.
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

JavaScript only:

Note: For the "name" parameter, the maximum string length is 512.
See Also

- FMOD_PLUGINTYPE
- System::getNumPlugins

Version 1.10.02 Built on Dec 7, 2017
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,
    int *systemrate,
    FMOD_SPEAKERMODE *speakermode,
    int *speakermodechannels,
    FMOD_DRIVER_STATE *state
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetRecordDriverInfo(
    FMOD_SYSTEM *system,
    int id,
    char *name,
    int namelen,
    FMOD_GUID *guid,
    int *systemrate,
    FMOD_SPEAKERMODE *speakermode,
    int *speakermodechannels,
    FMOD_DRIVER_STATE *state
);
```

C# Syntax

```csharp
RESULT System.getRecordDriverInfo(
    int id,
    StringBuilder name,
    int namelen,
    out Guid guid,
    out int systemrate,
    out SPEAKERMODE speakermode,
    out int speakermodechannels,
```
out DRIVER_STATE state
);

JavaScript Syntax

System.getRecordDriverInfo(
  id,
  name,  // writes value to name.val
  guid,  // writes value to guid.val
  systemrate,  // writes value to systemrate.val
  speakermode,  // writes value to speakermode.val
  speakermodechannels,  // writes value to speakermodechannels.val
  state  // writes value to state.val
);

Parameters

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

name
   Address of a variable that receives the name of the device encoded in a UTF-8 string. 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.

systemrate
   Address of a variable that receives the sample rate this device operates at. Optional. Specify 0 or NULL to ignore.

speakermode
   Address of a variable that receives the speaker setup this device is currently using. Optional. Specify 0 or NULL to ignore.

speakermodechannels
   Address of a variable that receives the number of channels in the current speaker setup. Optional. Specify 0 or NULL to ignore.

state
   Address of a variable that receives flags that provide additional information about the driver. 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

JavaScript only:

Note: For the "name" parameter, the maximum string length is 512.
See Also

- System::setOutput
- System::getRecordNumDrivers
Firelight Technologies FMOD Studio API
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,
    int *numconnected
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetRecordNumDrivers(
    FMOD_SYSTEM *system,
    int *numdrivers,
    int *numconnected
);
```

C# Syntax

```csharp
RESULT System.getRecordNumDrivers(
    out int numdrivers,
    out int numconnected
);
```

JavaScript Syntax

```javascript
System.getRecordNumDrivers(
    numdrivers, // writes value to numdrivers.val
    numconnected // writes value to numconnected.val
);
```
**Parameters**

numdrivers
   Address of a variable that receives the number of recording drivers available for this output mode. Optional. Specify 0 or NULL to ignore.

numconnected
   Address of a variable that receives the number of recording driver currently plugged in. 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::getRecordDriverInfo
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT System.getRecordPosition(
    int id,
    out uint position
);
```

**JavaScript Syntax**

```javascript
System.getRecordPosition(
    id,
    position // writes value to position.val
);
```
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.
Remarks

Will return `FMOD_ERR_RECORD_DISCONNECTED` if the driver is unplugged.

The position will return to 0 when `System::recordStop` is called or when a non-looping recording reaches the end.

*PS4 specific note*: Record devices are virtual so 'position' will continue to update if the device is unplugged (the OS is generating silence). This function will still report `FMOD_ERR_RECORD_DISCONNECTED` for your information though.
See Also

- System::recordStart
- System::recordStop
Firelight Technologies FMOD Studio API
System::getReverbProperties

Retrieves the current reverb environment for the specified reverb instance.

C++ Syntax

FMOD_RESULT System::getReverbProperties(
    int instance,
    FMOD_REVERB_PROPERTIES *prop
);

C Syntax

FMOD_RESULT FMOD_System_GetReverbProperties(
    FMOD_SYSTEM *system,
    int instance,
    FMOD_REVERB_PROPERTIES *prop
);

C# Syntax

RESULT System.getReverbProperties(
    int instance,
    out REVERB_PROPERTIES prop
);

JavaScript Syntax

System.getReverbProperties(
    instance,
    prop // writes value to prop.val
);
Parameters

instance
  Index of the particular reverb instance to target, from 0 to
  \texttt{FMOD\_REVERB\_MAXINSTANCES} inclusive.

prop
  Address of a variable that receives the current reverb environment
description.
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_REVERB_PROPERTIES**
- **System::setReverbProperties**
- **ChannelControl::setReverbProperties**
- **ChannelControl::getReverbProperties**
Firelight Technologies FMOD Studio API
System::getSoftwareChannels

Retrieves the maximum number of software mixed channels possible.

C++ Syntax

```cpp
FMOD_RESULT System::getSoftwareChannels(
    int *numsoftwarechannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_GetSoftwareChannels(
    FMOD_SYSTEM *system,
    int *numsoftwarechannels
);
```

C# Syntax

```csharp
RESULT System.getSoftwareChannels(
    out int numsoftwarechannels
);
```

JavaScript Syntax

```js
System.getSoftwareChannels( 
    numsoftwarechannels // writes value to numsoftwarecha
);
```
Parameters

numsoftwarechannels
    Address of a variable that receives the current maximum number of software voices available.
Return Values

If the function succeeds then the return 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 Studio API
System::getSoftwareFormat

Retrieves the output format for the software mixer.

C++ Syntax

FMOD_RESULT System::getSoftwareFormat(
   int *samplerate,
   FMOD_SPEAKERMODE *speakermode,
   int *numrawspeakers
);

C Syntax

FMOD_RESULT FMOD_System_GetSoftwareFormat(
   FMOD_SYSTEM *system,
   int *samplerate,
   FMOD_SPEAKERMODE *speakermode,
   int *numrawspeakers
);

C# Syntax

RESULT System.getSoftwareFormat(
   out int samplerate,
   out SPEAKERMODE speakermode,
   out int numrawspeakers
);

JavaScript Syntax

System.getSoftwareFormat(
   samplerate, // writes value to samplerate.val
   speakermode, // writes value to speakermode.va
   numrawspeakers // writes value to numrawspeakers
);
Parameters

samplerate
  Address of a variable that receives the sample rate of the mixer. Optional. Specify 0 or NULL to ignore.

speakermode
  Address of a variable that receives the speaker setup of the mixer. Optional. Specify 0 or NULL to ignore.

numrawspeakers
  Address of a variable that receives the number of speakers for **FMOD_SPEAKERMODE_RAW** mode. 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 require certain settings to initialize.
See Also

- System::setSoftwareFormat
- FMOD_SOUND_FORMAT
- FMOD_DSP_RESAMPLER
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

FMOD_RESULT System::getSoundRAM(
   int *currentallocated,
   int *maxallocated,
   int *total
);

C Syntax

FMOD_RESULT FMOD_System_GetSoundRAM(
   FMOD_SYSTEM *system,
   int *currentallocated,
   int *maxallocated,
   int *total
);

C# Syntax

RESULT System.getSoundRAM(
   out int currentallocated,
   out int maxallocated,
   out int total
);

JavaScript Syntax

System.getSoundRAM(
   currentallocated, // writes value to currentallocated
   maxallocated, // writes value to maxallocated.val
   total // writes value to total.val
);
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 Studio API
System::getSpeakerModeChannels

Gets the a speaker mode's channel count.

**C++ Syntax**

```cpp
FMOD_RESULT System::getSpeakerModeChannels(
    FMOD_SPEAKERMODE mode,
    int *channels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_GetSpeakerModeChannels(
    FMOD_SYSTEM *system,
    FMOD_SPEAKERMODE mode,
    int *channels
);
```

**C# Syntax**

```csharp
RESULT System.getSpeakerModeChannels(
    SPEAKERMODE mode,
    out int channels
);
```

**JavaScript Syntax**

```javascript
System.getSpeakerModeChannels(
    mode,
    channels // writes value to channels.val
);
```
Parameters

mode
channels
    Address of a variable that receives the number of channels.
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.
Firelight Technologies FMOD Studio API
System::getSpeakerPosition

Retrieves the current speaker position information for the selected speaker.

C++ Syntax

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

C Syntax

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

C# Syntax

```csharp
RESULT System.getSpeakerPosition(
    SPEAKER speaker,
    out float x,
    out float y,
    out bool active
);
```

JavaScript Syntax

```javascript
System.getSpeakerPosition(
    speaker,
    x,     // writes value to x.val
    y,     // writes value to y.val
    active // writes value to active.val
);
```
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::setSpeakerPosition` for more information on speaker positioning.
See Also

- System::setSpeakerPosition
- FMOD_SPEAKERMODE
- FMOD_SPEAKER
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT System.getStreamBufferSize(
    out uint filebuffersize,
    out TIMEUNIT filebuffersizetype
);
```

**JavaScript Syntax**

```javascript
System.getStreamBufferSize(
    filebuffersize, // writes value to filebuffersize
    filebuffersizetype // writes value to filebuffersize
);
```
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 Studio API
System::getUserData

Retrieves the user value 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
);
```

C# Syntax

```csharp
RESULT System.getUserData( 
    out IntPtr userdata
);
```

JavaScript Syntax

```javascript
System.getUserData( 
    userdata // writes value to userdata.val
);
```
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 Studio API
System::getVersion

Returns the current version of FMOD Studio 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
);
```

C# Syntax

```csharp
RESULT System.getVersion(
    out uint version
);
```

JavaScript Syntax

```javascript
System.getVersion(
    version // writes value to version.val
);
```
Parameters

version
   Address of a variable that receives the current FMOD Studio 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 Studio API
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 *extradriverdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_Init(
    FMOD_SYSTEM *system,
    int maxchannels,
    FMOD_INITFLAGS flags,
    void *extradriverdata
);
```

C# Syntax

```csharp
RESULT System.init(
    int maxchannels,
    INITFLAGS flags,
    IntPtr extradriverdata
);
```

JavaScript Syntax

```javascript
System.init(
    maxchannels,
    flags,
    extradriverdata
);
```
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 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.

extradrivertdata
   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 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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT System.isRecording(
    int id,
    out bool recording
);
```

**JavaScript Syntax**

```javascript
System.isRecording(
    id,
    recording // writes value to recording.val
);
```
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.

Will return FMOD_ERR_RECORD_DISCONNECTED if the driver is unplugged.

*PS4 specific note:* Record devices are virtual so 'recording' will continue to report true if the device is unplugged (the OS is generating silence). This function will still report FMOD_ERR_RECORD_DISCONNECTED for your information though.
See Also

- System::recordStart
- System::recordStop
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT System.loadGeometry(
    IntPtr data,
    int datasize,
    out Geometry geometry
);
```

**JavaScript Syntax**

```javascript
System.loadGeometry(
    data,
    datasize,
    geometry // writes value to geometry.val
);
```
**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 Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT System.loadPlugin(
    string filename,
    out uint handle
);
```

**JavaScript Syntax**

```javascript
System.loadPlugin(
);
```
Parameters

filename
Filename of the plugin to be loaded encoded in a UTF-8 string.

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

Plugins can be created for FMOD by the user, see the relevant section in the documentation on creating plugins.

Once the plugin is loaded, it can be enumerated and used. For file format plugins, FMOD will automatically try to use them during `System::createSound`. For DSP plugins, you can enumerate them with `System::getNumPlugins`, `System::getPluginHandle` and `System::getPluginInfo`.

The format of the plugin is dependant on the operating system:

- Windows uses .dll
- Linux uses .so
- Macintosh uses .dylib

**JavaScript only:**

This function is not currently supported
See Also

- System::setPluginPath
- System::unloadPlugin
- System::getNumPlugins
- System::getPluginHandle
- System::getPluginInfo
- System::setOutputByPlugin
- System::getOutputByPlugin
- System::createDSPByPlugin
- System::createSound
- FMOD_PLUGINTYPE
Firelight Technologies FMOD Studio API
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

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

C Syntax

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

C# Syntax

```csharp
RESULT System.lockDSP();
```

JavaScript Syntax

```javascript
System.lockDSP();
```
Return Values

If the function succeeds then the return value 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 Studio API
**System::mixerResume**

Resume mixer thread and reacquire access to audio hardware.

**C++ Syntax**

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

**C Syntax**

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

**C# Syntax**

```csharp
RESULT System.mixerResume();
```

**JavaScript Syntax**

```javascript
System.mixerResume();
```
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

Used on mobile platforms when entering the foreground after being suspended.

All internal state will resume, i.e. created sound and channels are still valid and playback will continue.

Android specific: Must be called on the same thread as System::mixerSuspend.
See Also

- System::mixerSuspend
Firelight Technologies FMOD Studio API
System::mixerSuspend

Suspend mixer thread and relinquish usage of audio hardware while maintaining internal state.

**C++ Syntax**

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

**C Syntax**

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

**C# Syntax**

```csharp
RESULT System.mixerSuspend();
```

**JavaScript Syntax**

```javascript
System.mixerSuspend();
```
**Return Values**

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

Used on mobile platforms when entering a backgrounded state to reduce CPU to 0%.

All internal state will be maintained, i.e. created sound and channels will stay available in memory.
See Also

- System::mixerResume
Firelight Technologies FMOD Studio API
System::playDSP

Plays a DSP unit object and its input network on a particular channel.

**C++ Syntax**

```cpp
FMOD_RESULT System::playDSP(
    FMOD::DSP *dsp,
    FMOD::ChannelGroup *channelgroup,
    bool paused,
    FMOD::Channel **channel
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_PlayDSP(
    FMOD_SYSTEM *system,
    FMOD_DSP *dsp,
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL paused,
    FMOD_CHANNEL **channel
);
```

**C# Syntax**

```csharp
RESULT System.playDSP(
    DSP dsp,
    ChannelGroup channelGroup,
    bool paused,
    out Channel channel
);
```

**JavaScript Syntax**

```javascript
System.playDSP(
    dsp,
    channelgroup,
    paused,
    channel       // writes value to channel.val
);
```
Parameters

dsp
Ptr to the DSP unit to play. This is opened with System::createDSP, System::createDSPByType, System::createDSPByPlugin.

channelgroup
Ptr to a channelgroup become a member of. This is more efficient than using Channel::setChannelGroup, as it does it during the channel setup, rather than connecting to the master channel group, then later disconnecting and connecting to the new channelgroup when specified. Optional. Use 0/NULL to ignore (use master ChannelGroup).

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
Ptr to a channel handle pointer that receives the newly playing channel. Optional. Use 0/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

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**.

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

- System::createDSP
- System::createDSPByType
- System::createDSPByPlugin
- Channel::setPaused
- Channel::setPriority
- System::init
Firelight Technologies FMOD Studio API
System::playSound

Plays a sound object on a particular channel and ChannelGroup if desired.

C++ Syntax

FMOD_RESULT System::playSound(
    FMOD::Sound *sound,
    FMOD::ChannelGroup *channelgroup,
    bool paused,
    FMOD::Channel **channel
);

C Syntax

FMOD_RESULT FMOD_System_PlaySound(
    FMOD_SYSTEM *system,
    FMOD_SOUND *sound,
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL paused,
    FMOD_CHANNEL **channel
);

C# Syntax

RESULT System.playSound(
    Sound sound,
    ChannelGroup channelGroup,
    bool paused,
    out Channel channel
);

JavaScript Syntax

System.playSound(
    sound,
    channelgroup,
    paused,
    channel
    // writes value to channel.val
);
Parameters

sound
  Pointer to the sound to play. This is opened with `System::createSound`.

channelgroup
  Pointer to a channelgroup become a member of. This is more efficient than using `Channel::setChannelGroup`, as it does it during the channel setup, rather than connecting to the master channel group, then later disconnecting and connecting to the new channelgroup when specified. Optional. Use 0/NULL to ignore (use master ChannelGroup).

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 unpasing with `Channel::setPaused` actually starts the sound.

channel
  Address of a channel handle pointer that receives the newly playing channel. Optional. Use 0/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

When a sound is played, it will use the sound's default frequency and priority.

A sound defined as FMOD_3D will by default play at the position of the listener. To set the 3D position of the channel before the sound is audible, start the channel paused by setting the paused flag to true, and calling Channel::set3DAttributes. Following that, unpause the channel with Channel::setPaused.

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

- System::createSound
- Channel::setPaused
- Channel::setPriority
- Channel::set3DAttributes
- Sound::setDefaults
- System::init
Firelight Technologies FMOD Studio API
System::recordStart

Starts the recording engine recording to the specified recording sound.

C++ Syntax

FMOD_RESULT System::recordStart(
    int id,
    FMOD::Sound *sound,
    bool loop
);

C Syntax

FMOD_RESULT FMOD_System_RecordStart(
    FMOD_SYSTEM *system,
    int id,
    FMOD_SOUND *sound,
    FMOD_BOOL loop
);

C# Syntax

RESULT System.recordStart(
    int id,
    Sound sound,
    bool loop
);

JavaScript Syntax

System.recordStart(
    id,
    sound,
    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.
Return Values

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

Will return `FMOD_ERR_RECORD_DISCONNECTED` if the driver is unplugged.

FMOD::Sound must be created as `FMOD_CREATESTREAM`.

Recording from the same driver a second time will stop the first recording.

For lowest latency set the FMOD::Sound sample rate to the rate returned by `System::getRecordDriverInfo`, otherwise a resampler will be allocated to handle the difference.
See Also

- System::recordStop
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.recordStop(
    int id
);
```

JavaScript Syntax

```javascript
System.recordStop(
    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 \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

Returns no error if unplugged or already stopped.
See Also

- System::recordStart
Firelight Technologies FMOD Studio API
System::registerCodec

Creates a file format codec to be used by FMOD for opening custom file types.

C++ Syntax

FMOD_RESULT System::registerCodec(
    FMOD_CODEC_DESCRIPTION *description,
    unsigned int *handle,
    unsigned int priority
);

C Syntax

FMOD_RESULT FMOD_System_RegisterCodec(
    FMOD_SYSTEM *system,
    FMOD_CODEC_DESCRIPTION *description,
    unsigned int *handle,
    unsigned int priority
);

C# Syntax

RESULT System.registerCodec(
    ref CODEC_DESCRIPTION description,
    out uint handle,
    uint priority
);

JavaScript Syntax

System.registerCodec(
    description,
    handle,
    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.
See Also

- `FMOD_CODEC_DESCRIPTION`
- `System::registerDSP`

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
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(
    const FMOD_DSP_DESCRIPTION *description,
    unsigned int *handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_RegisterDSP(
    FMOD_SYSTEM *system,
    const FMOD_DSP_DESCRIPTION *description,
    unsigned int *handle
);
```

C# Syntax

```csharp
RESULT System.registerDSP(
    ref DSP_DESCRIPTION description,
    out uint handle
);
```

JavaScript Syntax

```javascript
System.registerDSP(
    description,
    handle
);
```
Parameters

description
Address of an `FMOD_DSP_DESCRIPTION` structure, containing information about the DSP effect. Some members of `FMOD_DSP_DESCRIPTION` are referenced directly inside FMOD so the structure should be allocated statically or at least remain in memory for the lifetime of the system.

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**
- **System::registerCodec**
Firelight Technologies FMOD Studio API
System::registerOutput

Register a user-defined output mode for use with the System. This function allows you to register statically-linked output modes. Once registered, you can use the output mode with System::setOutputByPlugin.

C++ Syntax

```cpp
FMOD_RESULT System::registerOutput(
    const FMOD_OUTPUT_DESCRIPTION *description,
    unsigned int *handle
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_RegisterOutput(
    FMOD_SYSTEM *system,
    const FMOD_OUTPUT_DESCRIPTION *description,
    unsigned int *handle
);
```

C# Syntax

```csharp
RESULT System.registerOutput(    ref OUTPUT_DESCRIPTION description,    out uint handle    );
```

JavaScript Syntax

```javascript
System.registerOutput(    description,    handle    );
```
Parameters

description
   Address of an FMOD_OUTPUT_DESCRIPTION structure, containing information about the output mode.

description
   Address of a variable to receive the plugin handle of the newly-registered 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

- FMOD_OUTPUT_DESCRIPTION
- System::setOutputByPlugin
- System::getNumPlugins
- System::getPluginInfo

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
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);

C# Syntax

RESULT System.release();

JavaScript Syntax

System.release();
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 function also calls System::close, so calling close before this function is not necessary.

**NOTE!** Calls to System_Create and System::release are not thread-safe. Do not call these functions simultaneously from multiple threads at once.
See Also

- System_Create
- System::init
- System::close
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.set3DListenerAttributes(
   int listener,
   ref VECTOR pos,
   ref VECTOR vel,
   ref VECTOR forward,
   ref VECTOR up
);
```

JavaScript Syntax

```javascript
System.set3DListenerAttributes(
   listener,
   pos,
```
vel, forward, 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

Vectors should use your chosen coordinate system, see 3D sounds for more information.

Orientation vectors must be 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.

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.

NOTE! Users of the Studio API should call Studio::System::setListenerAttributes instead of this function.
See Also

- System::get3DListenerAttributes
- System::set3DSettings
- System::get3DSettings
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.set3DNumListeners(
    int numlisteners
);
```

JavaScript Syntax

```javascript
System.set3DNumListeners(
    numlisteners
);
```
Parameters

numlisteners
Number of listeners in the scene. Valid values are from 1 to \texttt{FMOD\_MAX\_LISTENERS} 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.

NOTE! Users of the Studio API should call `Studio::System::setNumListeners` instead of this function.
See Also

- System::get3DNumListeners
- System::set3DListenerAttributes
- FMOD_MAX_LISTENERS
Firelight Technologies FMOD Studio API
**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_ROLLOFF_CALLBACK callback);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_Set3DRolloffCallback(
    FMOD_SYSTEM *system,
    FMOD_3D_ROLLOFF_CALLBACK callback);
```

**C# Syntax**

```csharp
RESULT System.set3DRolloffCallback(
    CB_3D_ROLLOFFCALLBACK callback);
```

**JavaScript Syntax**

```javascript
System.set3DRolloffCallback(
    callback);
```
Parameters

callback
   Pointer to a C function of type `FMOD_3D_ROLLOFF_CALLBACK`, 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.
See Also

- **FMOD_3D_ROLLOFF_CALLBACK**
- **System::set3DListenerAttributes**
- **System::get3DListenerAttributes**
- **Channel::getUserData**
Firelight Technologies FMOD Studio API
System::set3DSettings

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

C++ Syntax

FMOD_RESULT System::set3DSettings(
    float dopplerscale,
    float distancefactor,
    float rolloffscale
);

C Syntax

FMOD_RESULT FMOD_System_Set3DSettings(
    FMOD_SYSTEM *system,
    float dopplerscale,
    float distancefactor,
    float rolloffscale
);

C# Syntax

RESULT System.set3DSettings(
    float dopplerscale,
    float distancefactor,
    float rolloffscale
);

JavaScript Syntax

System.set3DSettings(
    dopplerscale,
    distancefactor,
    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](https://www.sevenlogic.com/fmod/3dOUND/) based sounds only (which is the default). Volume for a sound set to [FMOD_3D_INVERSEROLLOFF](https://www.sevenlogic.com/fmod/3dOUND/) 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](https://www.sevenlogic.com/fmod/3dOUND/), [FMOD_3D_LINEARSQAROLLOFF](https://www.sevenlogic.com/fmod/3dOUND/) or [FMOD_3D_CUSTOMROLLOFF](https://www.sevenlogic.com/fmod/3dOUND/).
See Also

- System::get3DSettings
- Sound::set3DMinMaxDistance
- Sound::get3DMinMaxDistance
- Channel::set3DAttributes
- Channel::get3DAttributes
Firelight Technologies FMOD Studio API
System::setAdvancedSettings

Sets advanced features like configuring memory and cpu usage for FMOD_CREATECOMPRESSEDSAMPLE usage.

C++ Syntax

FMOD_RESULT System::setAdvancedSettings(
    FMOD_ADVANCEDSETTINGS *settings
);

C Syntax

FMOD_RESULT FMOD_System_SetAdvancedSettings(
    FMOD_SYSTEM *system,
    FMOD_ADVANCEDSETTINGS *settings
);

C# Syntax

RESULT System.setAdvancedSettings(
    ref ADVANCEDSETTINGS settings
);

JavaScript Syntax

System.setAdvancedSettings(
    settings
);
Parameters

settings
  Pointer to FMOD_ADVANCEDSETTINGS structure.
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_ADVANCEDSETTINGS
- System::getAdvancedSettings
- FMOD_MODE
Firelight Technologies FMOD Studio API
System::setCallback

Sets a system callback to catch various fatal or informational events.

C++ Syntax

```cpp
FMOD_RESULT System::setCallback(
    FMOD_SYSTEM_CALLBACK callback,
    FMOD_SYSTEM_CALLBACK_TYPE callbackmask
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetCallback(
    FMOD_SYSTEM *system,
    FMOD_SYSTEM_CALLBACK callback,
    FMOD_SYSTEM_CALLBACK_TYPE callbackmask
);
```

C# Syntax

```csharp
RESULT System.setCallback(
    SYSTEM_CALLBACK callback,
    SYSTEM_CALLBACK_TYPE callbackmask
);
```

JavaScript Syntax

```javascript
System.setCallback(
    callback,
    callbackmask
);
```
Parameters

callback
   Pointer to a callback to receive the event callback when it happens.

callbackmask
   A bitfield describing which callbacks are required to be triggered. Masking out some callback types can help avoid a flood of irrelevant callbacks being triggered.
Return Values

If the function succeeds then the return value 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 function name.

The 'userdata' parameter passed to the callback is the userdata assigned to the system from System::setUserData function.

Example:

```c
FMOD_RESULT F_CALLBACK systemcallback(FMOD_SYSTEM *system, FMOD_SYSTEM_CALLBACK_TYPE type)
{
    FMOD::System *sys = (FMOD::System *)system;

    switch (type)
    {
    case FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED:
    {
        int numdrivers;
        printf("NOTE : FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED occurred.\n");
        sys->getNumDrivers(&numdrivers);
        printf("Numdevices = %d\n", numdrivers);
        break;
    }
    case FMOD_SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED:
    {
        printf("ERROR : FMOD_SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED\n";
        printf("%s\n", commanddata1);
        printf("%d bytes.\n", commanddata2);
        break;
    }
    case FMOD_SYSTEM_CALLBACK_THREADCREATED:
    {
        printf("NOTE : FMOD_SYSTEM_CALLBACK_THREADCREATED occurred.\n";
        printf("Thread ID = %d\n", (int)commanddata1);
        printf("Thread Name = %s\n", (char *)commanddata2);
        break;
    }
    case FMOD_SYSTEM_CALLBACK_BADDSPCONNECTION:
```
{  
  FMOD::DSP *source = (FMOD::DSP *)commanddata1;
  FMOD::DSP *dest = (FMOD::DSP *)commanddata2;

  printf("ERROR : %s\n", FMOD_SYSTEM_CALLBACK_BADDSPCONNECTION);
  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_CALLBACK_BADDSPLEVEL:
{
  printf("NOTE : %s\n", FMOD_SYSTEM_CALLBACK_BADDSPLEVEL);
  break;
}

case FMOD_SYSTEM_CALLBACK_PREMIX:
{
  printf("NOTE : %s\n", FMOD_SYSTEM_CALLBACK_PREMIX);
  break;
}

case FMOD_SYSTEM_CALLBACK_MIDMIX:
{
  printf("NOTE : %s\n", FMOD_SYSTEM_CALLBACK_MIDMIX);
  break;
}

case FMOD_SYSTEM_CALLBACK_POSTMIX:
{
  printf("NOTE : %s\n", FMOD_SYSTEM_CALLBACK_POSTMIX);
  break;
}

return FMOD_OK;
}
See Also

- System::update
- FMOD_SYSTEM_CALLBACK
- FMOD_SYSTEM_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
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/unstable 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
);
```

**C# Syntax**

```csharp
RESULT System.setDSPBufferSize(
    uint bufferlength,
    int numbuffers
);
```

**JavaScript Syntax**

```javascript
System.setDSPBufferSize(
    bufferlength,
    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 numblocks;
float ms;

result = system->getDSPBufferSize(&blocksize, &numblocks);
result = system->getSoftwareFormat(&frequency, 0, 0);
ms = (float)blocksize * 1000.0f / (float)frequency;

printf("Mixer blocksize = %.02f ms\n", ms);
printf("Mixer Total buffersize = %.02f ms\n", ms * numbblocks);
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::getDSPBuffer` after calling `System::init` to see if this is the case.
- 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
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT System.setDriver(
    int driver
);
```

**JavaScript Syntax**

```javascript
System.setDriver(
    driver
);
```
Parameters

driver
   Driver number to select. 0 = primary or main sound device as selected by the operating system settings. Use System::getNumDrivers and System::getDriverInfo to determine available devices.
Return Values

If the function succeeds then the return value 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.

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 Studio API
System::setFileSystem

Specify user callbacks for FMOD's internal file manipulation functions. This function is useful for replacing FMOD's file system with a game system's own file reading API.

**C++ Syntax**

```cpp
FMOD_RESULT System::setFileSystem(
    FMOD_FILE_OPEN_CALLBACK useropen,
    FMOD_FILE_CLOSE_CALLBACK userclose,
    FMOD_FILE_READ_CALLBACK userread,
    FMOD_FILE_SEEK_CALLBACK userseek,
    FMOD_FILEASYNCREAD_CALLBACK userasyncread,
    FMOD_FILEASYNCCANCEL_CALLBACK userasynccancel,
    int blockalign
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_System_SetFileSystem(
    FMOD_SYSTEM *system,
    FMOD_FILE_OPEN_CALLBACK useropen,
    FMOD_FILE_CLOSE_CALLBACK userclose,
    FMOD_FILE_READ_CALLBACK userread,
    FMOD_FILE_SEEK_CALLBACK userseek,
    FMOD_FILEASYNCREAD_CALLBACK userasyncread,
    FMOD_FILEASYNCCANCEL_CALLBACK userasynccancel,
    int blockalign
);
```

**C# Syntax**

```csharp
RESULT System.setFileSystem(
    FILE_OPENCALLBACK useropen,
    FILE_CLOSECALLBACK userclose,
    FILE_READCALLBACK userread,
    FILE_SEEKCALLBACK userseek,
    FILEASYNCREADCALLBACK userasyncread,
    FILEASYNCCANCELCALLBACK userasynccancel,
    int blockalign
);
```
JavaScript Syntax

System.setFileSystem(
    useropen,
    userclose,
    userread,
    userseek,
    userasyncread,
    userasynccancel,
    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 recommended 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_xxxCALLBACK 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/userasyncccanel).
- 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_OPEN_CALLBACK
- FMOD_FILE_CLOSE_CALLBACK
- FMOD_FILE_READ_CALLBACK
- FMOD_FILE_SEEK_CALLBACK
- FMOD_FILEASYNCREAD_CALLBACK
- FMOD_FILEASYNCCANCEL_CALLBACK

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.setGeometrySettings(
    float maxworldsize
);
```

JavaScript Syntax

```javascript
System.setGeometrySettings(
    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
System::setNetworkProxy

Set a proxy server to use for all subsequent internet connections.

C++ Syntax

FMOD_RESULT System::setNetworkProxy(
    const char *proxy
);

C Syntax

FMOD_RESULT FMOD_System_SetNetworkProxy(
    FMOD_SYSTEM *system,
    const char *proxy
);

C# Syntax

RESULT System.setNetworkProxy(
    string proxy
);

JavaScript Syntax

System.setNetworkProxy(
    proxy
);
Parameters

proxy
The name of a proxy server encoded as a UTF-8 string 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 Studio API
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
);
```

C# Syntax

```csharp
RESULT System.setNetworkTimeout(
    int timeout
);
```

JavaScript Syntax

```javascript
System.setNetworkTimeout(
    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.
Remarks

The default timeout is 5000ms.
See Also

- System::getNetworkTimeout
Firelight Technologies FMOD Studio API
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
);```

**C# Syntax**

```csharp
RESULT System.setOutput(
    OUTPUTTYPE output
);```

**JavaScript Syntax**

```javascript
System.setOutput(
    output
);```
Parameters

output
Output type to select. See FMOD_OUTPUTTYPE 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 only necessary 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.

(Windows Only) This function can be called after FMOD is already activated, you can use it to change the output mode at runtime. If \texttt{FMOD\_SYSTEM\_CALLBACK\_DEVICELISTCHANGED} is specified use the \texttt{setOutput} call to change to \texttt{FMOD\_OUTPUTTYPE\_NOSOUND} if no more sound card drivers exist.

\textbf{NOTE!} When using the Studio API, switching to an NRT output mode after FMOD is already activated will not behave correctly unless the Studio API was initialized with \texttt{FMOD\_STUDIO\_INIT\_SYNCHRONOUS\_UPDATE}. 
See Also

- System::init
- System::close
- System::getOutput
- FMOD_OUTPUTTYPE
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT System.setOutputByPlugin(
    uint handle
);
```

**JavaScript Syntax**

```javascript
System.setOutputByPlugin(
    handle
);
```
Parameters

handle
   Handle to a pre-existing output plugin.
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

(Windows Only) This function can be called after FMOD is already activated. You can use it to change the output mode at runtime. If `FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED` is specified use the setOutput call to change to `FMOD_OUTPUTTYPE_NOSOUND` if no more sound card drivers exist.
See Also

- System::getNumPlugins
- System::getOutputByPlugin
- System::setOutput
- System::init
- System::close
Firelight Technologies FMOD Studio API
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**

FMOD_RESULT System::setPluginPath(  
    const char *path  
);

**C Syntax**

FMOD_RESULT FMOD_System_SetPluginPath(  
    FMOD_SYSTEM *system,  
    const char *path  
);

**C# Syntax**

RESULT System.setPluginPath(  
    string path  
);

**JavaScript Syntax**

System.setPluginPath(  
    path  
);
Parameters

path
   A character string containing a correctly formatted path to load plugins from encoded in a UTF-8 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::loadPlugin
- System::init
Firelight Technologies FMOD Studio API
System::setReverbProperties

Sets parameters for the global reverb environment.

To assist in defining reverb properties there are several presets available, see FMOD_REVERB_PRESETS

C++ Syntax

FMOD_RESULT System::setReverbProperties(
    int instance,
    const FMOD_REVERB_PROPERTIES *prop
);

C Syntax

FMOD_RESULT FMOD_System_SetReverbProperties(
    FMOD_SYSTEM *system,
    int instance,
    const FMOD_REVERB_PROPERTIES *prop
);

C# Syntax

RESULT System.setReverbProperties(
    int instance,
    ref REVERB_PROPERTIES prop
);

JavaScript Syntax

System.setReverbProperties(
    instance,
    prop
);
**Parameters**

**instance**
Index of the particular reverb instance to target, from 0 to **FMOD_REVERB_MAXINSTANCES** inclusive.

**prop**
Address of an **FMOD_REVERB_PROPERTIES** structure which defines the attributes for the reverb. Passing 0 or NULL to this function will delete the physical 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

When using each instance for the first time, FMOD will create a physical SFX reverb DSP unit that takes up several hundred kilobytes of memory and some CPU.
See Also

- `FMOD_REVERB_PROPERTIES`
- `FMOD_REVERB_PRESETS`
- `System::getReverbProperties`
- `ChannelControl::setReverbProperties`
- `ChannelControl::getReverbProperties`
Firelight Technologies FMOD Studio API
System::setSoftwareChannels

Sets the maximum number of software mixed channels possible.

C++ Syntax

```cpp
FMOD_RESULT System::setSoftwareChannels(
   int numsoftwarechannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetSoftwareChannels(
   FMOD_SYSTEM *system,
   int numsoftwarechannels
);
```

C# Syntax

```csharp
RESULT System.setSoftwareChannels(
   int numsoftwarechannels
);
```

JavaScript Syntax

```javascript
System.setSoftwareChannels(
   numsoftwarechannels
);
```
**Parameters**

numsoftwarechannels
The maximum number of mixable voices to be allocated by FMOD, default = 64.
**Return Values**

If the function succeeds then the return value 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, it must be called before `System::init`, or after `System::close`. 
See Also

- System::getSoftwareChannels
- System::init
- System::close
Firelight Technologies FMOD Studio API
System::setSoftwareFormat

Sets the output format for the software mixer.

If loading Studio banks, this must be called with speakermode corresponding to the project's output format if there is a possibility of the output audio device not matching the project's format. Any differences between the project format and the system's speakermode will cause the mix to sound wrong.

If not loading Studio banks, do not call this unless you explicitly want to change a setting from the default. FMOD will default to the speaker mode and sample rate that the OS / output prefers.

C++ Syntax

```cpp
FMOD_RESULT System::setSoftwareFormat(
    int samplerate,
    FMOD_SPEAKERMODE speakermode,
    int numrawspeakers
);
```

C Syntax

```c
FMOD_RESULT FMOD_System_SetSoftwareFormat(
    FMOD_SYSTEM *system,
    int samplerate,
    FMOD_SPEAKERMODE speakermode,
    int numrawspeakers
);
```

C# Syntax

```csharp
RESULT System.setSoftwareFormat(
    int samplerate,
    SPEAKERMODE speakermode,
    int numrawspeakers
);
```

JavaScript Syntax

---
System.setSoftwareFormat(
    samplerate,
    speakermode,
    numrawspeakers
);
Parameters

samplerate
Sample rate in Hz, that the software mixer will run at. Specify values between 8000 and 192000.

speakermode
Speaker setup for the software mixer.

numrawspeakers
Number of output channels / speakers to initialize the sound card to in \texttt{FMOD\_SPEAKERMODE\_RAW} mode. Optional. Specify 0 to ignore. Maximum of \texttt{FMOD\_MAX\_CHANNEL\_WIDTH}.
**Return Values**

If the function succeeds then the return value 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, it must be called before `System::init`, or after `System::close`. 
See Also

- System::getSoftwareFormat
- FMOD_SPEAKERMODE
- System::init
- System::close
- FMOD_MAX_CHANNEL_WIDTH
Firelight Technologies FMOD Studio API
**System::setSpeakerPosition**

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 funtion 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**

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

**C Syntax**

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

**C# Syntax**

```csharp
RESULT System.setSpeakerPosition(
    SPEAKER speaker,
    float x,
    float y,
    bool active
);
```

**JavaScript Syntax**
System.setSpeakerPosition(
    speaker,
    x,
    y,
    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.

y
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 \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

A typical stereo setup would look like this.

```cpp
system->setSpeakerPosition(FMOD_SPEAKER_FRONT_LEFT, -1.0f, 0.0f,
system->setSpeakerPosition(FMOD_SPEAKER_FRONT_RIGHT, 1.0f, 0.0f,
A typical 7.1 setup would look like this.

```cpp
system->setSpeakerPosition(FMOD_SPEAKER_FRONT_LEFT, sin(degtorad system->setSpeakerPosition(FMOD_SPEAKER_FRONT_RIGHT, sin(degtorad system->setSpeakerPosition(FMOD_SPEAKER_FRONT_CENTER, sin(degtorad system->setSpeakerPosition(FMOD_SPEAKER_LOW_FREQUENCY, sin(degtorad system->setSpeakerPosition(FMOD_SPEAKER_SURROUND_LEFT, sin(degtorad system->setSpeakerPosition(FMOD_SPEAKER_SURROUND_RIGHT, sin(degtorad system->setSpeakerPosition(FMOD_SPEAKER_REAR_LEFT, sin(degtorad system->setSpeakerPosition(FMOD_SPEAKER_REAR_RIGHT, sin(degtorad

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_REAR_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::setSoftwareFormat` overrides these values, so this function must be called after this.
See Also

- System::getSpeakerPosition
- System::setSoftwareFormat
- FMOD_SPEAKERMODE
- FMOD_SPEAKER
Firelight Technologies FMOD Studio API
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 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
);
```

C# Syntax

```csharp
RESULT System.setStreamBufferSize(
    uint filebuffersize,
    TIMEUNIT filebuffersizetype
);
```

JavaScript Syntax

```javascript
System.setStreamBufferSize(
    filebuffersize,
    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_CREATESOUNDEXINFO`.
See Also

- **FMOD_TIMEUNIT**
- **System::createSound**
- **System::getStreamBufferSize**
- **Sound::getOpenState**
- **Channel::setMute**
- **Memory_GetStats**
- **FMOD_CREATESOUNDEXINFO**
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.setUserData(
    IntPtr userdata
);
```

JavaScript Syntax

```javascript
System.setUserData(
    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
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT System.unloadPlugin(
    uint handle
);
```

JavaScript Syntax

```javascript
System.unloadPlugin(
    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 Studio API
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);
```

**C# Syntax**

```csharp
RESULT System.unlockDSP();
```

**JavaScript Syntax**

```javascript
System.unlockDSP();
```
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

The DSP engine must be locked with `System::lockDSP` before this function is called.
See Also

- System::lockDSP
Firelight Technologies FMOD Studio API
System::update

Updates the FMOD system. This should be called once per 'game' tick, or once per frame in your application.

C++ Syntax

FMOD_RESULT System::update();

C Syntax

FMOD_RESULT FMOD_System_Update(FMOD_SYSTEM *system);

C# Syntax

RESULT System.update();

JavaScript Syntax

System.update();
Return Values

If the function succeeds then the return value 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 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.

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.
See Also

- System::init
- FMOD_INITFLAGS
- FMOD_OUTPUTTYPE
- FMOD_MODE
Firelight Technologies FMOD Studio API
Sound
Functions

Sound::addSyncPoint
Sound::deleteSyncPoint
Sound::get3DConeSettings
Sound::get3DCustomRolloff
Sound::get3DMinMaxDistance
Sound::getDefaults
Sound::getFormat
Sound::getLength
Sound::getLoopCount
Sound::getLoopPoints
Sound::getMode
Sound::getMusicChannelVolume
Sound::getMusicNumChannels
Sound::getMusicSpeed
Sound::getName
Sound::getNumSubSounds
Sound::getNumSyncPoints
Sound::getNumTags
Sound::getOpenState
Sound::getSoundGroup
Sound::getSubSound
Sound::getSubSoundParent
Sound::getSyncPoint
Sound::getSyncPointInfo
Sound::getSystemObject
Sound::getTag
Sound::getUserData
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::setUserData
Sound::unlock
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT Sound.addSyncPoint(
    uint offset,
    TIMEUNIT offsettype,
    string name,
    out IntPtr point
);
```

**JavaScript Syntax**

```javascript
Sound.addSyncPoint(
);
```
Parameters

offset
  offset in units specified by offsettype to add the callback syncpoint for a sound.

offsettype
  offset type to describe the offset provided. Could be PCM samples or milliseconds for example.

name
  A name character string to be stored with the sync point. This will be provided via the sync point callback.

point
  Address of a variable to store a pointer to the newly created syncpoint.
Return Values

If the function succeeds then the return value 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.

JavaScript only:

This function is not currently supported
See Also

- `Sound::getNumSyncPoints`
- `Sound::getSyncPoint`
- `Sound::getSyncPointInfo`
- `Sound::deleteSyncPoint`
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.deleteSyncPoint(
    IntPtr point
);
```

**JavaScript Syntax**

```javascript
Sound.deleteSyncPoint(
);
```
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.

JavaScript only:

This function is not currently supported
See Also

- `Sound::getNumSyncPoints`
- `Sound::getSyncPoint`
- `Sound::getSyncPointInfo`
- `Sound::addSyncPoint`
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT Sound.get3DConesettings(
    out float insideconeangle,
    out float outsideconeangle,
    out float outsidevolume
);
```

JavaScript Syntax

```javascript
Sound.get3DConesettings(
    insideconeangle, // writes value to insideconeangle
    outsideconeangle, // writes value to outsideconeangle
    outsidevolume    // writes value to 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 \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

- `Sound::set3DConeSettings`
- `Channel::set3DConeSettings`
Firelight Technologies FMOD Studio API
Sound::get3DCustomRolloff

Retrieves a pointer to the sound’s current custom rolloff curve.

C++ Syntax

FMOD_RESULT Sound::get3DCustomRolloff(
    FMOD_VECTOR **points,
    int *numpoints
);

C Syntax

FMOD_RESULT FMOD_Sound_Get3DCustomRolloff(
    FMOD_SOUND *sound,
    FMOD_VECTOR **points,
    int *numpoints
);

C# Syntax

RESULT Sound.get3DCustomRolloff(
    out IntPtr points,
    out int numpoints
);

JavaScript Syntax

Sound.get3DCustomRolloff(
    points,                       // writes value to points.val
    numpoints                     // writes value to numpoints.val
);
**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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.get3DMinMaxDistance(
    out float min,
    out float max
);
```

**JavaScript Syntax**

```javascript
Sound.get3DMinMaxDistance( 
    min, // writes value to min.val
    max // writes value to max.val
);
```
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
**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,
    int *priority
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_GetDefaults(
    FMOD_SOUND *sound,
    float *frequency,
    int *priority
);
```

**C# Syntax**

```csharp
RESULT Sound.getDefaults(
    out float frequency,
    out int priority
);
```

**JavaScript Syntax**

```javascript
Sound.getDefaults(
    frequency,          // writes value to frequency.val
    priority            // writes value to priority.val
);
```
**Parameters**

**frequency**
Address of a variable that receives the default frequency for the sound. 
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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Sound.getFormat(
    out SOUND_TYPE type,
    out SOUND_FORMAT format,
    out int channels,
    out int bits
);
```

JavaScript Syntax

```javascript
Sound.getFormat(  
    type, // writes value to type.val  
    format, // writes value to format.val  
    channels, // writes value to channels.val  
    bits // writes value to bits.val
);
```
**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. 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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Sound.getLength(
    out uint length,
    TIMEUNIT lengthtype
);
```

JavaScript Syntax

```javascript
Sound.getLength(
    length, // writes value to length.val
    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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Sound.getLoopCount(
    out int loopcount
);
```

JavaScript Syntax

```javascript
Sound.getLoopCount(
    loopcount // writes value to loopcount.val
);
```
**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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.getLoopPoints(
    out uint loopstart,
    TIMEUNIT loopstarttype,
    out uint loopend,
    TIMEUNIT loopendtype
);
```

**JavaScript Syntax**

```javascript
Sound.getLoopPoints(
    loopstart,                   // writes value to loopstart.val
    loopstarttype,
    loopend,                     // writes value to loopend.val
    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::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
);
```

C# Syntax

```csharp
RESULT Sound.getMode(
    out MODE mode
);
```

JavaScript Syntax

```javascript
Sound.getMode(
    mode // writes value to mode.val
);
```
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 \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

- `Sound::setMode`
- `System::createSound`
- `Channel::setMode`
- `Channel::getMode`
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT Sound.getMusicChannelVolume(
    int channel,
    out float volume
);
```

**JavaScript Syntax**

```javascript
Sound.getMusicChannelVolume(
    channel,
    volume // writes value to volume.val
);
```
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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Sound.getMusicNumChannels(
    out int numchannels
);
```

JavaScript Syntax

```javascript
Sound.getMusicNumChannels(
    numchannels // writes value to numchannels.value
);
```
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 Studio API
Sound::getMusicSpeed

Gets 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
);
```

C# Syntax

```csharp
RESULT Sound.getMusicSpeed(
   out float speed
);
```

JavaScript Syntax

```javascript
Sound.getMusicSpeed( 
   speed // writes value to speed.val
);
```
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 Studio API
Sound::getName

Retrieves the name of a sound.

C++ Syntax

FMOD_RESULT Sound::getName(
    char *name,
    int namelen
);

C Syntax

FMOD_RESULT FMOD_Sound_GetName(
    FMOD_SOUND *sound,
    char *name,
    int namelen
);

C# Syntax

RESULT Sound.getName(
    StringBuilder name,
    int namelen
);

JavaScript Syntax

Sound.getName(
    name
    // writes value to name.val
);
Parameters

name
   Address of a variable that receives the name of the sound encoded in a UTF-8 string.

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)".

JavaScript only:

Note: For the "name" parameter, the maximum string length is 512.
See Also

- System::createSound
- FMOD_MODE
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.getNumSubSounds(
    out int numsubsounds
);
```

**JavaScript Syntax**

```javascript
Sound.getNumSubSounds(
    numsubsounds // writes value to numsubsounds.v
);
```
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 Studio API
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

FMOD_RESULT Sound::getNumSyncPoints(
    int *numsyncpoints
);

C Syntax

FMOD_RESULT FMOD_Sound_GetNumSyncPoints(
    FMOD_SOUND *sound,
    int *numsyncpoints
);

C# Syntax

RESULT Sound.getNumSyncPoints(
    out int numsyncpoints
);

JavaScript Syntax

Sound.getNumSyncPoints(
    numsyncpoints // writes value to 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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Sound.getNumTags(
    out int numtags,
    out int numtagsupdated
);
```

JavaScript Syntax

```javascript
Sound.getNumTags(
    numtags,          // writes value to numtags.val
    numtagsupdated    // writes value to 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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Sound.getOpenState(
    out OPENSTATE openstate,
    out uint percentbuffered,
    out bool starving,
    out bool diskbusy
);
```

JavaScript Syntax

```javascript
Sound.getOpenState(  
    openstate, // writes value to openstate.val  
    percentbuffered, // writes value to percentbuffered  
    starving, // writes value to starving.val  
    diskbusy // writes value to diskbusy.val
);
```
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.

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 **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 opened with `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 `Channel::setMute` to keep it quiet until it is not starving any more.

**Note:** Always check 'openstate' to determine the state of the sound. Do not assume that if this function returns `FMOD_OK` then the sound has finished loading.
See Also

- **FMOD_OPENSTATE**
- **FMOD_MODE**
- **Channel::setMute**
Firelight Technologies FMOD Studio API
**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_SOUNDGROUP **soundgroup
);
```

**C# Syntax**

```csharp
RESULT Sound.getSoundGroup(  
    out SoundGroup soundgroup
);
```

**JavaScript Syntax**

```javascript
Sound.getSoundGroup(  
    soundgroup  // writes value to soundgroup.val
);
```
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.
See Also

- `Sound::setSoundGroup`
- `System::getMasterSoundGroup`
Firelight Technologies FMOD Studio API
Sound::getSubSound

Retrieves a handle to a Sound object that is contained within the parent sound.

C++ Syntax

FMOD_RESULT Sound::getSubSound(
    int index,
    FMOD::Sound **subsound
);

C Syntax

FMOD_RESULT FMOD_Sound_GetSubSound(
    FMOD_SOUND *sound,
    int index,
    FMOD_SOUND **subsound
);

C# Syntax

RESULT Sound.getSubSound(
    int index,
    out Sound subsound
);

JavaScript Syntax

Sound.getSubSound(
    index,
    subsound               // writes value to subsound.val
);
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 \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

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::getSubSoundParent
- System::createSound
- FMOD_MODE
- FMOD_OPENSTATE
Firelight Technologies FMOD Studio API
**Sound::getSubSoundParent**

Retrieves a handle to the parent Sound object that contains our subsound.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::getSubSoundParent(
    FMOD::Sound **parentsound
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_GetSubSoundParent(
    FMOD_SOUND *sound,
    FMOD_SOUND **parentsound
);
```

**C# Syntax**

```csharp
RESULT Sound.getSubSoundParent(
    out Sound parentsound
);
```

**JavaScript Syntax**

```javascript
Sound.getSubSoundParent(
    parentsound // writes value to parentsound.va
);
```
Parameters

dparents

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 not a subsound, the parentsound will be returned as NULL.
See Also

- `Sound::getNumSubSounds`
- `Sound::getSubSound`
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.getSyncPoint(
    int index,
    out IntPtr point
);
```

**JavaScript Syntax**

```javascript
Sound.getSyncPoint(
);
```
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.

JavaScript only :

This function is not currently supported
See Also

- `Sound::getNumSyncPoints`
- `Sound::getSyncPointInfo`
- `Sound::addSyncPoint`
- `Sound::deleteSyncPoint`
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT Sound.getSyncPointInfo(
    IntPtr point,
    StringBuilder name,
    int namelen,
    out uint offset,
    TIMEUNIT offsettype
);
```

**JavaScript Syntax**

```javascript
Sound.getSyncPointInfo(
);
```
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 \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

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.

**JavaScript only :**

This function is not currently supported
See Also

- `Sound::getNumSyncPoints`
- `Sound::getSyncPoint`
- `Sound::addSyncPoint`
- `Sound::deleteSyncPoint`
Firelight Technologies FMOD Studio API
**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);
```

**C# Syntax**

```csharp
RESULT Sound.getSystemObject(
    out System system);
```

**JavaScript Syntax**

```javascript
Sound.getSystemObject(
    system // writes value to system.val
);
```
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 Studio API
Sound::getTag

Retrieves a descriptive tag stored by the sound, to describe things like the song name, author etc.

C++ Syntax

FMOD_RESULT Sound::getTag(
    const char *name,
    int index,
    FMOD_TAG *tag
);

C Syntax

FMOD_RESULT FMOD_Sound_GetTag(
    FMOD_SOUND *sound,
    const char *name,
    int index,
    FMOD_TAG *tag
);

C# Syntax

RESULT Sound.getTag(
    string name,
    int index,
    out TAG tag
);

JavaScript Syntax

Sound.getTag(
    name,
    index,
    tag // writes value to tag.val
);
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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.getUserData(
    out IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
Sound.getUserData(
    userdata // writes value to userdata.val
);
```
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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Sound.lock(
    uint offset,
    uint length,
    out IntPtr ptr1,
    out IntPtr ptr2,
    out uint len1,
    out uint len2
);
```

JavaScript Syntax

```javascript
Sound.lock(
    offset,
```
length,
ptr1,       // writes value to ptr1.val
ptr2,       // writes value to ptr2.val
len1,       // writes value to len1.val
len2        // writes value to len2.val
);
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 \texttt{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. You must take this into consideration when processing the data within the pointer.
See Also

- **Sound::unlock**
- **System::createSound**
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.readData(
    IntPtr buffer,
    uint length,
    out uint read
);
```

**JavaScript Syntax**

```javascript
Sound.readData(
    buffer,   // writes value to buffer.val
    length,
    read,     // writes value to read.val
);
```
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 Studio API
**Sound::release**

Frees a sound object.

**C++ Syntax**

```cpp
FMOD_RESULT Sound::release();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_Release(FMOD_SOUND *sound);
```

**C# Syntax**

```csharp
RESULT Sound.release();
```

**JavaScript Syntax**

```javascript
Sound.release();
```
Return Values

If the function succeeds then the return value 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, 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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.seekData(
    uint pcm
);
```

**JavaScript Syntax**

```javascript
Sound.seekData(
    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 Studio API
Sound::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

FMOD_RESULT Sound::set3DConeSettings(
   float insideconeangle,
   float outsideconeangle,
   float outsidevolume
);

C Syntax

FMOD_RESULT FMOD_Sound_Set3DConeSettings(
   FMOD_SOUND *sound,
   float insideconeangle,
   float outsideconeangle,
   float outsidevolume
);

C# Syntax

RESULT Sound.set3DConeSettings(
   float insideconeangle,
   float outsideconeangle,
   float outsidevolume
);

JavaScript Syntax

Sound.set3DConeSettings(
   insideconeangle,
   outsideconeangle,
   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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.set3DCustomRolloff(
    ref VECTOR points,
    int numpoints
)
```

**JavaScript Syntax**

```javascript
Sound.set3DCustomRolloff(
    points,
    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.

```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**
- **Sound::get3DCustomRolloff**
- **Channel::set3DCustomRolloff**
- **Channel::get3DCustomRolloff**
Firelight Technologies FMOD Studio API
**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 don't 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,
    float min,
```
float max
);

C# Syntax

RESULT Sound.set3DMinMaxDistance(
    float min,
    float max
);

JavaScript Syntax

Sound.set3DMinMaxDistance(
    min,
    max
);
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::set3DSets`
Firelight Technologies FMOD Studio API
**Sound::setDefaults**

Sets a sounds'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,
    int priority
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Sound_SetDefaults(
    FMOD_SOUND *sound,
    float frequency,
    int priority
);
```

**C# Syntax**

```csharp
RESULT Sound.setDefaults(
    float frequency,
    int priority
);
```

**JavaScript Syntax**

```javascript
Sound.setDefaults(
    frequency,
    priority
);
```
**Parameters**

**frequency**
Default playback frequency for the sound, in hz. (ie 44100hz).

**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 Studio API
Sound::setLoopCount

Sets a sound, by default, to loop a specified number of times before stopping if its mode is set to \texttt{FMOD\_LOOP\_NORMAL} or \texttt{FMOD\_LOOP\_BIDI}.

\textbf{C++ Syntax}

\begin{verbatim}
FMOD\_RESULT Sound::setLoopCount(
    int loopcount
);
\end{verbatim}

\textbf{C Syntax}

\begin{verbatim}
FMOD\_RESULT FMOD\_Sound\_SetLoopCount(
    FMOD\_SOUND *sound,
    int loopcount
);
\end{verbatim}

\textbf{C# Syntax}

\begin{verbatim}
RESULT Sound.setLoopCount(
    int loopcount
);
\end{verbatim}

\textbf{JavaScript Syntax}

\begin{verbatim}
Sound.setLoopCount(
    loopcount
);
\end{verbatim}
**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

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

- Sound::getLoopCount
- System::setStreamBufferSize
- FMOD_CREATE_SOUNDEXINFO
Firelight Technologies FMOD Studio API
Sound::setLoopPoints

Sets the loop points within a sound.

C++ Syntax

```c++
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
);
```

C# Syntax

```csharp
RESULT Sound.setLoopPoints(
    uint loopstart,
    TIMEUNIT loopstarttype,
    uint loopend,
    TIMEUNIT loopendtype
);
```

JavaScript Syntax

```javascript
Sound.setLoopPoints(
    loopstart,
    loopstarttype,
    loopend,
    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**

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 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 loopp points 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 Studio API
Sound::setMode

Sets or alters the mode of a sound.

C++ Syntax

FMOD_RESULT Sound::setMode(
    FMOD_MODE mode
);

C Syntax

FMOD_RESULT FMOD_Sound_SetMode(
    FMOD_SOUND *sound,
    FMOD_MODE mode
);

C# Syntax

RESULT Sound.setMode(
    MODE mode
);

JavaScript Syntax

Sound.setMode(
    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 \texttt{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:

- \texttt{FMOD\_LOOP\_OFF}
- \texttt{FMOD\_LOOP\_NORMAL}
- \texttt{FMOD\_LOOP\_BIDI}
- \texttt{FMOD\_3D\_HEADRELATIVE}
- \texttt{FMOD\_3D\_WORLDRELATIVE}
- \texttt{FMOD\_2D}
- \texttt{FMOD\_3D}
- \texttt{FMOD\_3D\_INVERSEROLLOFF}
- \texttt{FMOD\_3D\_LINEARROLLOFF}
- \texttt{FMOD\_3D\_LINEARSQUAREROLLOFF}
- \texttt{FMOD\_3D\_CUSTOMROLLOFF}
- \texttt{FMOD\_3D\_IGNOREGEOMETRY}

Issues with streamed audio. (Sounds created with with \texttt{System::createStream} or \texttt{FMOD\_CREATESTREAM}).

When changing the loop mode, sounds created with \texttt{System::createStream} or \texttt{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 \texttt{FMOD\_CREATESOUNDEXINFO}.

If this happens, you may need to reflush the stream buffer. To do this, you can call \texttt{Channel::setPosition} which forces a reflush 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 \texttt{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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Sound.setMusicChannelVolume(
    int channel,
    float volume
);
```

**JavaScript Syntax**

```javascript
Sound.setMusicChannelVolume(
    channel,
    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 Studio API
Sound::setMusicSpeed

Sets the relative speed of MOD/S3M/XM/IT/MIDI music.

C++ Syntax

FMOD_RESULT Sound::setMusicSpeed(
    float speed
);

C Syntax

FMOD_RESULT FMOD_Sound_SetMusicSpeed(
    FMOD_SOUND *sound,
    float speed
);

C# Syntax

RESULT Sound.setMusicSpeed(
    float speed
);

JavaScript Syntax

Sound.setMusicSpeed(
    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 Studio API
**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_SOUNDDGROUP *soundgroup
);
```

**C# Syntax**

```csharp
RESULT Sound.setSoundGroup(
    SoundGroup soundgroup
);
```

**JavaScript Syntax**

```javascript
Sound.setSoundGroup(
    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`
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT Sound.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
Sound.setUserData(
    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 Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT Sound.unlock(
    IntPtr ptr1,
    IntPtr ptr2,
    uint len1,
    uint len2
);
```

**JavaScript Syntax**

```javascript
Sound.unlock(
    ptr1,
    ptr2,
    len1,
    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.
See Also

- **Sound::lock**
- **System::createSound**
Firelight Technologies FMOD Studio API
ChannelControl

The base class for both Channels and Channel Groups.
Functions

ChannelControl::addDSP
ChannelControl::addFadePoint
ChannelControl::get3DAttributes
ChannelControl::get3DAttributes
ChannelControl::get3DConeOrientation
ChannelControl::get3DConeSettings
ChannelControl::get3DCustomRolloff
ChannelControl::get3DDistanceFilter
ChannelControl::get3DDopplerLevel
ChannelControl::get3DLevel
ChannelControl::get3DMinMaxDistance
ChannelControl::get3DOcclusion
ChannelControl::get3DSpread
ChannelControl::getAudibility
ChannelControl::getDSP
ChannelControl::getDSPClock
ChannelControl::getDSPIndex
ChannelControl::getDelay
ChannelControl::getFadePoints
ChannelControl::getLowPassGain
ChannelControl::getMixMatrix
ChannelControl::getMode
ChannelControl::getMute
ChannelControl::getNumDSPs
ChannelControl::getPaused
ChannelControl::getPitch
ChannelControl::getReverbProperties
ChannelControl::getSystemObject
ChannelControl::getUserData
ChannelControl::getVolume
ChannelControl::getVolumeRamp
ChannelControl::isPlaying
ChannelControl::removeDSP
ChannelControl::removeFadePoints
ChannelControl::set3DAttributes
ChannelControl::set3DConeOrientation
ChannelControl::set3DConeSettings
ChannelControl::set3DCustomRolloff
ChannelControl::set3DDistanceFilter
ChannelControl::set3DDopplerLevel
ChannelControl::set3DLevel
ChannelControl::set3DMinMaxDistance
ChannelControl::set3DOcclusion
ChannelControl::set3DSpread
ChannelControl::setCallback
ChannelControl::setDSPIndex
ChannelControl::setDelay
ChannelControl::setFadePointRamp
ChannelControl::setLowPassGain
ChannelControl::setMixLevelsInput
ChannelControl::setMixLevelsOutput
ChannelControl::setMixMatrix
ChannelControl::setMode
ChannelControl::setMute
ChannelControl::setPan
ChannelControl::setPaused
ChannelControl::setPitch
ChannelControl::setReverbProperties
ChannelControl::setUserData
ChannelControl::setVolume
ChannelControl::setVolumeRamp
ChannelControl::stop
Firelight Technologies FMOD Studio API
ChannelControl::addDSP

Add a pre-created DSP unit to the specified index in the DSP chain.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::addDSP(
    int index,
    DSP *dsp
);
```
Parameters

\textit{index}

Offset to add this DSP unit at in the DSP chain, see \texttt{FMOD\_CHANNEL\_CONTROL\_DSP\_INDEX} for special named offsets.

\textit{dsp}

Pointer to a pre-created DSP unit to be inserted at the specified offset.
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

- System::createDSP
- System::createDSPByType
- System::createDSPByPlugin
- ChannelControl::removeDSP
- ChannelControl::getDSP
- ChannelControl::getNumDSPs
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
ChannelControl::addFadePoint

Add a volume point to fade from or towards, using a clock offset and 0 to 1 volume level.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::addFadePoint(
    unsigned long long dspclock,
    float volume
);
```
Parameters

\textit{dspclock}

DSP clock of the parent channel group to set the fade point volume.

\textit{volume}

Volume level where 0 is silent and 1.0 is normal volume. Amplification is supported.
Return Values

If the function succeeds then the return value 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 every fade point, FMOD will do a per sample volume ramp between them. It will scale with the current Channel or ChannelGroup's volume.

// Ramp from full volume to half volume over the next 4096 samples
FMOD_RESULT result;
unsigned long long parentclock;
result = target->getDSPClock(NULL, &parentclock);
result = target->addFadePoint(parentclock, 1.0f);
result = target->addFadePoint(parentclock + 4096, 0.5f);
See Also

- ChannelControl::removeFadePoints
- ChannelControl::setFadePointRamp
- ChannelControl::getFadePoints
Firelight Technologies FMOD Studio API
ChannelControl::get3DAttributes

Retrieves the position and velocity used to apply panning, attenuation and doppler.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::get3DAttributes(
    FMOD_VECTOR *pos,
    FMOD_VECTOR *vel,
    FMOD_VECTOR *alt_pan_pos
);
```
Parameters

pos

Address of a variable that receives the position in 3D space used for panning and attenuation. Optional, specify 0 or NULL to ignore.

vel

Address of a variable that receives the velocity in 'distance units per second' (see remarks) in 3D space. Optional, specify 0 or NULL to ignore.

alt_pan_pos

(Unimplemented).
Return Values

If the function succeeds then the return value 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

- [ChannelControl::set3DAttributes](#)
- [System::set3DSettings](#)
- [FMOD_VECTOR](#)
Firelight Technologies FMOD Studio API
ChannelControl::get3D ConeOrientation

Retrieves the orientation of the sound projection cone.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::get3D ConeOrientation(
    FMOD_VECTOR *orientation
);
```
Parameters

*orientation*

Address of a variable that receives the coordinates of the sound cone orientation vector, 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

- ChannelControl::set3DConeOrientation
Firelight Technologies FMOD Studio API
ChannelControl::get3DConesSettings

Retrieves the angles that define the sound projection cone including the volume when outside the cone.

C++ Syntax

FMOD_RESULT ChannelControl::get3DConesSettings(
    float *insideconeangle,
    float *outsideconeangle,
    float *outsidevolume
);
**Parameters**

*insideconeangle*

Address of a variable that receives the inside cone angle, in degrees. This is the angle within which the sound is at its normal volume.

*outsideconeangle*

Address of a variable that receives the outside cone angle, in degrees. This is the angle outside of which the sound is at its outside volume.

*outsidevolume*

Address of a variable that receives the cone outside 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

- [ChannelControl::set3DConeSettings](#)
Firelight Technologies FMOD Studio API
ChannelControl::get3DCustomRolloff

Retrieves a pointer to the current custom rolloff curve.

C++ Syntax

FMOD_RESULT ChannelControl::get3DCustomRolloff(
    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

- ChannelControl::set3DCustomRolloff
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
ChannelControl::get3DDistanceFilter

Retrieve the settings for the 3D distance filter properties for a Channel or Channel Group.

C++ Syntax

FMOD_RESULT ChannelControl::get3DDistanceFilter(
    bool *custom,
    float *customLevel,
    float *centerFreq
);
Parameters

*custom*

Address of a variable to receive the enabled/disabled state of the FMOD distance rolloff calculation. Default = false.

*customLevel*

Address of a variable to receive the manual user attenuation, where 1.0 = no attenuation and 0 = complete attenuation. Default = 1.0.

*centerFreq*

Address of a variable to receive center frequency in hz for the high-pass filter used to simulate distance attenuation, from 10.0 to 22050.0. Default = 1500.0.
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

- ChannelControl::set3DDistanceFilter
ChannelControl::get3DDopplerLevel

Retrieves the amount by which doppler is scaled.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::get3DDopplerLevel(
    float *level
);
```
Parameters

*level*

Address of a variable to receive the doppler scale from 0.0 (none), to 1.0 (normal) to 5.0 (exaggerated).
Return Values

If the function succeeds then the return 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

- ChannelControl::set3DDopplerLevel
Firelight Technologies FMOD Studio API
ChannelControl::get3DLevel

Retrieves the current 3D mix level set by ChannelControl::set3DPanLevel.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::get3DLevel(  
    float *level  
);
```
**Parameters**

*level*

3D pan level from 0.0 (attenuation is ignored and panning as set by 2D panning functions) to 1.0 (pan and attenuate according to 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

- ChannelControl::set3DPanLevel
Firelight Technologies FMOD Studio API
ChannelControl::get3DMinMaxDistance

Retrieves the minimum and maximum audible distance.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::get3DMinMaxDistance(
    float *mindistance,
    float *maxdistance
);
```
Parameters

mindistance

Address of a variable that receives the minimum volume distance in 'units' (see remarks). Optional, specify 0 or NULL to ignore.

maxdistance

Address of a variable that receives the maximum volume distance in 'units' (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

- ChannelControl::set3DMinMaxDistance
- System::set3DSettings
Firelight Technologies FMOD Studio API
ChannelControl::get3DOcclusion

Retrieves the occlusion factors.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::get3DOcclusion(
    float *directocclusion,
    float *reverbocclusion
);
```
Parameters

*directocclusion*

Address of a variable that receives the occlusion factor for the direct path.

*reverbocclusion*

Address of a variable that receives the occlusion factor for the reverb mix.
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

- ChannelControl::set3DOcclusion
Firelight Technologies FMOD Studio API
ChannelControl::get3DSpread

Retrieves the spread of a 3D sound in speaker space.

C++ Syntax

FMOD_RESULT ChannelControl::get3DSpread(
    float *angle
);
Parameters

angle

Address of a variable that receives the speaker spread angle.
**Return Values**

If the function succeeds then the return 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

- [ChannelControl::set3DSpread](#)
Firelight Technologies FMOD Studio API
ChannelControl::getAudibility

Retrieves the combined volume after 3D spatialization and geometry occlusion calculations including any volumes set via the API.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getAudibility(  
    float *audibility
);
```
Parameters

audibility

Address of a variable that receives the 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 result of all volume modifiers. This value is used by the virtual channel system to order its channels between real and virtual.

See the Virtual Voice System page for more details about how the audibility is calculated.
See Also

- `Channel::isVirtual`
- `ChannelControl::getVolume`
- `ChannelControl::get3DOcclusion`
- `ChannelControl::get3DAttributes`
- `FMOD_DSP_PARAMETER_OVERALLGAIN`
- `FMOD_DSP_PARAMETER_DATA_TYPE_OVERALLGAIN`
Firelight Technologies FMOD Studio API
ChannelControl::getDSP

Retrieve the DSP unit at the specified index.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getDSP(
    int index,
    DSP **dsp
);
```
Parameters

\textit{index}

Offset into the DSP chain, see \texttt{FMOD\_CHANNELCONTROL\_DSP\_INDEX} for special named offsets.

\textit{dsp}

Address of a variable to receive a pointer to the requested 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.
See Also

- ChannelControl::addDSP
- ChannelControl::removeDSP
- ChannelControl::getNumDSPs
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
ChannelControl::getDSPClock

Retrieves the DSP clock values which count up by the number of samples per second in the software mixer, i.e. if the default sample rate is 48KHz, the DSP clock increments by 48000 per second.

C++ Syntax

FMOD_RESULT ChannelControl::getDSPClock(
    unsigned long long *dspclock,
    unsigned long long *parentclock
);
Parameters

dspclock
Address of a variable to receive the DSP clock value for the head DSP node.

parentclock
Address of a variable to receive the DSP clock value for the tail DSP node.
Return Values

If the function succeeds then the return value 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 `ChannelControl::setDelay` to play a sound on an exact tick in the future, or stop it in the future.

Note that when delaying a channel or channel group you want to sync it to the parent channel group DSP clock value, not its own DSP clock value.
See Also

- `ChannelControl::setDelay`
- `ChannelControl::getDelay`
Firelight Technologies FMOD Studio API
ChannelControl::getDSPIndex

Retrieves the index in the DSP chain of the provided DSP.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getDSPIndex(
    DSP *dsp,
    int *index
);
```
Parameters

\textit{dsp}

Pointer to a DSP unit that exists in the DSP chain.

\textit{index}

Address of a variable to receive the offset in the DSP chain of the specified DSP.
Return Values

If the function succeeds then the return 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

- ChannelControl::setDSPIndex
Firelight Technologies FMOD Studio API
ChannelControl::getDelay

Retrieves a start (and/or stop) time relative to the parent channel group DSP clock, with sample accuracy.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getDelay(
    unsigned long long *dspclock_start,
    unsigned long long *dspclock_end,
    bool *stopchannels
);
```
Parameters

dspclock_start

Address of a variable that receives the DSP clock of the parent channel group to audibly start playing sound at. Optional, specify 0 or NULL to ignore.

dspclock_end

Address of a variable that receives the DSP clock of the parent channel group to audibly stop playing sound at. Optional, specify 0 or NULL to ignore.

stopchannels

Address of a variable that receives TRUE = stop according to ChannelControl::isPlaying. FALSE = remain 'active' and a new start delay could start playback again at a later time. Optional, specify 0 or NULL to ignore.
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

- `ChannelControl::setDelay`
- `ChannelControl::getDSPClock`
- `ChannelControl::isPlaying`
Firelight Technologies FMOD Studio API
ChannelControl::getFadePoints

Retrieve information about fade points stored within a Channel or ChannelGroup.

C++ Syntax

FMOD_RESULT ChannelControl::getFadePoints(
    unsigned int *numpoints,
    unsigned long long *point_dspclock,
    float *point_volume
);
**Parameters**

*numpoints*

Address of a variable to receive the number of fade points stored within the Channel or ChannelGroup.

*point_dspclock*

Address of a variable to receive an array of 64bit clock values. Can be 0 or NULL.

*point_volume*

Address of a variable to receive an array of floating point volume values. Can be 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 `FMOD_RESULT` enumeration.
Remarks

To first get the number of points for memory purposes, and not store any data, call this function with point_dpsclock and point_volume parameters being 0 or NULL.
See Also

- ChannelControl::addFadePoint
- ChannelControl::removeFadePoints
Firelight Technologies FMOD Studio API
ChannelControl::getLowPassGain

Retrieves the gain of the dry signal when lowpass filtering is applied.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getLowPassGain(
    float *gain
);
```
Parameters

gain

Address of a variable that receives the linear gain level, from 0 (silent) to 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

- `ChannelControl::setLowPassGain`
Firelight Technologies FMOD Studio API
ChannelControl::getMixMatrix

Retrieves a 2D pan matrix that maps input channels (columns) to output speakers (rows).

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getMixMatrix(
    float *matrix,
    int *outchannels,
    int *inchannels,
    int matrixhop
);
```
Parameters

*matrix*

Address of a 2 dimensional array of volume levels in row-major order. Each row represents an output speaker, each column represents an input channel.

*outchannels*

Address of a variable to receive the number of output channels (rows) in the matrix being passed in.

*inchannels*

Address of a variable to receive the number of input channels (columns) in the matrix being passed in.

*matrixhop*

The width (total number of columns) of the matrix. Optional. If this is 0, inchannels will be taken as the width of the matrix. Maximum of `FMOD_MAX_CHANNEL_WIDTH`.
Return Values

If the function succeeds then the return value 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 gain for input channel 's' to output channel 't' is matrix[t * matrixhop + s].

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

The matrix size will generally be the size of the number of channels in the current speaker mode. Use System::getSoftwareFormat to determine this.

Passing NULL for 'matrix' will allow you to query 'outchannels' and 'inchannels' without copying any data.
See Also

- `ChannelControl::setMixMatrix`
- `System::getSoftwareFormat`
- `FMOD_MAX_CHANNEL_WIDTH`
Firelight Technologies FMOD Studio API
ChannelControl::getMode

Retrieves the mode bit flags for the channel.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getMode(
    FMOD_MODE *mode
);
```
Parameters

mode

Address of a variable to receive the mode bits.
Return Values

If the function succeeds then the return 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
- FMOD_MODE
ChannelControl::getMute

Retrieves the mute state.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getMute(
    bool *mute
);
```
Parameters

mute

Address of a variable that receives the current mute state, true = mute (silent), false = normal volume.
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

- ChannelControl::setMute
Firelight Technologies FMOD Studio API
ChannelControl::getNumDSPs

Retrieves the number of DSP units in the DSP chain.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getNumDSPs(int *numdsps);
```
Parameters

\textit{numdps}

Address of a variable that receives the number of DSP units in the DSP chain.
Return Values

If the function succeeds then the return 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

- `ChannelControl::addDSP`
- `ChannelControl::removeDSP`
- `ChannelControl::getDSP`
Firelight Technologies FMOD Studio API
ChannelControl::getPaused

Retrieves the paused state.

C++ Syntax

```c++
FMOD_RESULT ChannelControl::getPaused(
    bool *paused
);
```
Parameters

paused

Address of a variable that receives the current paused state, true = paused, false = 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

- ChannelControl::setPaused
Firelight Technologies FMOD Studio API
ChannelControl::getPitch

Retrieves the pitch value.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getPitch(
    float *pitch
);
```
Parameters

pitch

Address of a variable to receive the pitch value, 0.5 = half pitch, 2.0 = double pitch, etc.
Return Values

If the function succeeds then the return 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

- ChannelControl::setPitch
- Channel::setFrequency
- Channel::getFrequency
Firelight Technologies FMOD Studio API
ChannelControl::getReverbProperties

Retrieves the wet level (or send level) for a particular reverb instance.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getReverbProperties(
    int instance,
    float *wet
);
```
Parameters

instance

Index of the particular reverb instance to target, from 0 to
**FMOD_REVERB_MAXINSTANCES** inclusive.

wet

Address of a variable that receives the send level for the signal to the reverb,
from 0 (none) to 1.0 (full).
Return Values

If the function succeeds then the return 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

- ChannelControl::setReverbProperties
Firelight Technologies FMOD Studio API
ChannelControl::getSystemObject

Retrieves the parent System object that created the channel or channel group.

C++ Syntax

```c++
FMOD_RESULT ChannelControl::getSystemObject(
    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
- System::playSound
Firelight Technologies FMOD Studio API
ChannelControl::getUserData

Retrieves a user value that can be set with ChannelControl::setUserData.

C++ Syntax

FMOD_RESULT ChannelControl::getUserData(
    void **userdata
);

Parameters

`userdata`

Address of a variable to receive data that the user has stored 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.
See Also

- ChannelControl::setUserData
Firelight Technologies FMOD Studio API
ChannelControl::getVolume

Retrieves the volume level.

C++ Syntax

FMOD_RESULT ChannelControl::getVolume(
    float *volume
);

Parameters

volume

Address of a variable to receive the linear volume 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.
See Also

- ChannelControl::setVolume
Firelight Technologies FMOD Studio API
ChannelControl::getVolumeRamp

Retrieves whether volume ramp is enabled.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::getVolumeRamp(
    bool *ramp
);
```
Parameters

*ramp*

Address of a variable to receive the volume ramp 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

- ChannelControl::setVolumeRamp
Firelight Technologies FMOD Studio API
ChannelControl::isPlaying

Retrieves the playing state.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::isPlaying(
    bool *isplaying
);
```
Parameters

isplaying

Address of a variable that receives the current playing status, true = currently playing a sound, false = not playing a 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.
See Also

- System::playSound
- System::playDSP
Firelight Technologies FMOD Studio API
ChannelControl::removeDSP

Remove a particular DSP unit from the DSP chain.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::removeDSP(
    DSP *dsp
);
```
Parameters

dsp

Pointer to a DSP unit (that exists in the DSP chain) you wish to remove.
Return Values

If the function succeeds then the return 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

- `ChannelControl::addDSP`
- `ChannelControl::getDSP`
- `ChannelControl::getNumDSPs`
Firelight Technologies FMOD Studio API
ChannelControl::removeFadePoints

Remove volume fade points on the timeline. This function will remove multiple fade points with a single call if the points lay between the 2 specified clock values (inclusive).

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::removeFadePoints(
    unsigned long long dspclock_start,
    unsigned long long dspclock_end
);
```
Parameters

\textit{dspclock\_start}

DSP clock of the parent channel group to start removing fade points from.

\textit{dspclock\_end}

DSP clock of the parent channel group to start removing fade points 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.
See Also

- ChannelControl::addFadePoint
- ChannelControl::getFadePoints
Firelight Technologies FMOD Studio API
ChannelControl::set3DAttributes

Sets the position and velocity used to apply panning, attenuation and doppler.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::set3DAttributes(
    const FMOD_VECTOR *pos,
    const FMOD_VECTOR *vel,
    const FMOD_VECTOR *alt_pan_pos
);
```
Parameters

*pos*

Position in 3D space used for panning and attenuation. Optional, specify 0 or NULL to ignore.

*vel*

Velocity in 'distance units per second' (see remarks) in 3D space. Optional, specify 0 or NULL to ignore.

*alt_pan_pos*

(Unimplemented).
Return Values

If the function succeeds then the return value 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 ChannelControl::set3DSpread.

Vectors should use your chosen coordinate system, see 3D sounds for more information.
See Also

- `ChannelControl::get3DAttributes`
- `ChannelControl::set3DSpread`
- `System::set3DSettings`
- `FMOD_VECTOR`
Firelight Technologies FMOD Studio API
ChannelControl::set3DConEOrientation

Sets the orientation of the sound projection cone.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::set3DConEOrientation(
    FMOD_VECTOR *orientation
);
```
Parameters

orientation

Coordinates of the sound cone orientation vector, 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.
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

- ChannelControl::get3DConeOrientation
- ChannelControl::set3DConeSettings
- Sound::set3DConeSettings
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
ChannelControl::set3DConesSettings

Sets the angles that define the sound projection cone including the volume when outside the cone.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::set3DConesSettings(
    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 0.0 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

To define the parameters per sound use `Sound::set3D ConeSettings`. 
See Also

- ChannelControl::get3DConeSettings
- ChannelControl::set3DConeOrientation
- Sound::set3DConeSettings
Firelight Technologies FMOD Studio API
ChannelControl::set3DCustomRolloff

Sets a custom rolloff curve to define how audio will attenuate over distance. Must be used in conjunction with `FMOD_3D_CUSTOMROLLOFF` flag to be activated.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelControl::set3DCustomRolloff(
    FMOD_VECTOR *points,
    int numpoints
);
```
Parameters

*points*

Array of `FMOD_VECTOR` structures where `x` = distance and `y` = volume from 0.0 to 1.0. `z` should be set to 0.

*numpoints*

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.

Values set with ChannelControl::setMinMaxDistance are meaningless when **FMOD_3D_CUSTOMROLLOFF** is used, their values are ignored.

Here is an example of a custom array of points.

```c
static FMOD_VECTOR curve[3] =
{
    { 0.0f, 1.0f, 0.0f },
    { 2.0f, 0.2f, 0.0f },
    { 20.0f, 0.0f, 0.0f }
};
```

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.

To define the parameters per sound use **Sound::set3DCustomRolloff**.
See Also

- ChannelControl::get3DCustomRolloff
- ChannelControl::setMinMaxDistance
- Sound::set3DCustomRolloff
- FMOD_VECTOR
- FMOD_3D_CUSTOMROLLOFF
Firelight Technologies FMOD Studio API
ChannelControl::set3DDistanceFilter

Control the behaviour of a 3D distance filter, whether to enable or disable it, and frequency characteristics.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::set3DDistanceFilter(
    bool custom,
    float customLevel,
    float centerFreq
);
```
Parameters

custom

Specify true to disable FMOD distance rolloff calculation. Default = false.

customLevel

Specify a attenuation factor manually here, where 1.0 = no attenuation and 0 = complete attenuation. Default = 1.0.

centerFreq

Specify a center frequency in hz for the high-pass filter used to simulate distance attenuation, from 10.0 to 22050.0. Default = 1500.0.
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

- ChannelControl::get3DDistanceFilter
Firelight Technologies FMOD Studio API
ChannelControl::set3DDopplerLevel

Sets the amount by which doppler is scaled.

C++ Syntax

FMOD_RESULT ChannelControl::set3DDopplerLevel(
    float level
);
Parameters

level

Doppler scale from 0.0 (none), to 1.0 (normal) to 5.0 (exaggerated), 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

- ChannelControl::get3DDopplerLevel
Firelight Technologies FMOD Studio API
ChannelControl::set3DLevel

Sets how much the 3D engine has an effect on the channel, versus that set by 2D panning functions.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::set3DLevel(
    float level
);
```
Parameters

*level*

3D pan level from 0.0 (attenuation is ignored and panning as set by 2D panning functions) to 1.0 (pan and attenuate according to 3D position), default = 1.0.
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

Only affects sounds created FMOD_3D.

2D panning functions include ChannelControl::setPan, ChannelControl::setMixLevelsOutput, ChannelControl::setMixLevelsInput, ChannelControl::setMixMatrix, etc

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).
See Also

- ChannelControl::get3DLevel
- ChannelControl::setPan
- ChannelControl::setMixLevelsOutput
- ChannelControl::setMixLevelsInput
- ChannelControl::setMixMatrix
Firelight Technologies FMOD Studio API
ChannelControl::set3DMinMaxDistance

Sets the minimum and maximum audible distance.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::set3DMinMaxDistance(
    float mindistance,
    float maxdistance
);
```
Parameters

mindistance
Minimum volume distance in 'units' (see remarks), default = 1.0.

maxdistance
Maximum volume distance in 'units' (see remarks), default = 10000.0.
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 the listener is in-between the minimum distance and the sound source the volume will be at its maximum. As the listener moves from the minimum distance to the maximum distance the sound will attenuate following the rolloff curve set. When outside the maximum distance the sound will no longer attenuate.

Minimum distance 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 small mindistance such as 0.1. This 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 causing the volume to stay at its loudest until the listener was 100 meters away, then it would be hundreds of meters more before it would fade out.

Maximum distance 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 don't need to. Some people have the confusion that maxdistance is the point the sound will fade out to zero, 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.

To define the min and max distance per sound use `Sound::set3DMinMaxDistance`.

If `FMOD_3D_CUSTOMROLLOFF` is used, then these values are stored, but ignored in 3D processing.
See Also

- [ChannelControl::get3DMinMaxDistance](#)
- [System::set3DSettings](#)
- [Sound::set3DMinMaxDistance](#)
- [FMOD_3D_CUSTOMROLLOFF](#)

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
ChannelControl::set3DOcclusion

Sets the occlusion factors manually for when the FMOD geometry engine is not being used.

C++ Syntax

FMOD_RESULT ChannelControl::set3DOcclusion(
    float directocclusion,
    float reverbocclusion
);
Parameters

**directocclusion**

Occlusion factor for the direct path, from 0.0 (not occluded) to 1.0 (fully occluded), default = 0.0.

**reverbocclusion**

Occlusion factor for the reverb mix, from 0.0 (not occluded) to 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

Normally the volume is simply attenuated by the 'directocclusion' factor however if `FMOD_INIT_CHANNEL_LOWPASS` is specified frequency filtering will be used with a very small CPU hit.
See Also

- ChannelControl::get3DOcclusion
- FMOD_INIT_CHANNEL_LOWPASS
Firelight Technologies FMOD Studio API
ChannelControl::set3DSpread

Sets the spread of a 3D sound in speaker space.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::set3DSpread(
    float angle
);
```
Parameters

angle

Speaker spread angle. 0 = all sound channels are located at the same speaker location and is 'mono'. 360 = all sound channels 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

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'.

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 channel located 15 degrees apart from each other in the speaker array.

Mono sounds are spread as if they were a stereo signal, i.e. the signal is split into 2. The power will remain the same as it spreads 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 stero 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!
See Also

- ChannelControl::get3DSpread
Firelight Technologies FMOD Studio API
ChannelControl::setCallback

Sets a callback to perform action for a specific event.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setCallback(
    FMOD_CHANNELCONTROL_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 driven by `System::update` and will only occur when this function is called. This has the main advantage of far less complication due to thread issues, and allows all FMOD commands, including loading sounds and playing new sounds from the callback. It also allows any type of sound to have an end callback, no matter what it is. The only disadvantage is that 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` in between your return type and function name.

```c
FMOD_RESULT F_CALLBACK mycallback(FMOD_CHANNELCONTROL *chanControl,
{
    if (controlType == FMOD_CHANNELCONTROL_TYPE_CHANNEL)
    {
        FMOD::Channel *channel = (FMOD::Channel *)chanControl;
        // Channel specific functions here...
    }
    else
    {
        FMOD::ChannelGroup *group = (FMOD::ChannelGroup *)chanControl;
        // ChannelGroup specific functions here...
    }
    // ChannelControl generic functions here...
    return FMOD_OK;
}
```
See Also

- System::update
- FMOD_CHANNELCONTROL_CALLBACK
- FMOD_CHANNELCONTROL_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
ChannelControl::setDSPIndex

Moves the position in the DSP chain of a specified DSP unit.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setDSPIndex(
    DSP *dsp,
    int index
);```
Parameters

`dsp`

Pointer to a DSP unit that exists in the DSP chain.

`index`

Offset in the DSP chain to move the DSP to, see `FMOD_CHANNELCONTROL_DSP_INDEX` for special named offsets.
Return Values

If the function succeeds then the return value 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 reordering DSP units inside a Channel or ChannelGroup so that processing can happen in the desired order.

You can verify the order of the DSP chain using iteration via `ChannelControl::getNumDSPs` and `ChannelControl::getDSP` or with the FMOD Profiler tool.
See Also

- ChannelControl::getDSPIndex
- ChannelControl::getNumDSPs
- ChannelControl::getDSP
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
ChannelControl::setDelay

Sets a start (and/or stop) time relative to the parent channel group DSP clock, with sample accuracy.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setDelay(
    unsigned long long dspclock_start,
    unsigned long long dspclock_end,
    bool stopchannels
);
```
Parameters

* dspclock_start

DSP clock of the parent channel group to audibly start playing sound at, a value of 0 indicates no delay.

* dspclock_end

DSP clock of the parent channel group to audibly stop playing sound at, a value of 0 indicates no delay.

* stopchannels

TRUE = stop according to `ChannelControl::isPlaying`. FALSE = remain 'active' and a new start delay could start playback again at a later 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

Every channel and channel group has its own DSP Clock. A channel or channel group can be delayed relatively against its parent, with sample accurate positioning. To delay a sound, use the 'parent' channel group DSP clock to reference against when passing values into this function.

If a parent channel group changes its pitch, the start and stop times will still be correct as the parent clock is rate adjusted by that pitch.
See Also

- `ChannelControl::getDelay`
- `ChannelControl::getDSPClock`
- `ChannelControl::isPlaying`
Firelight Technologies FMOD Studio API
ChannelControl::setFadePointRamp

Add a short 64 sample volume ramp to the specified time in the future using fade points.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setFadePointRamp(
    unsigned long long dspclock,
    float volume
);
```
Parameters

*dspclock*

DSP clock of the parent channel group when the volume will be ramped to.

*volume*

Volume level where 0 is silent and 1.0 is normal volume. Amplification is supported.
Return Values

If the function succeeds then the return value 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 helper function that automatically ramps from the current fade volume to the newly provided volume at a specified time. It will clear any fade points set after this time. Use in conjunction with `ChannelControl::setDelay` stop delay, to ramp down volume before it stops. The user would specify the same clock value for the fade ramp and stop delay. This can also be used as a way to provide sample accurate delayed volume changes without clicks.
See Also

- ChannelControl::setDelay
- ChannelControl::removeFadePoints
- ChannelControl::addFadePoint
- ChannelControl::getFadePoints
Firelight Technologies FMOD Studio API
ChannelControl::setLowPassGain

Sets the gain of the dry signal when lowpass filtering is applied.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setLowPassGain(
    float gain
);
```
Parameters

gain

Linear gain level, from 0 (silent, full filtering) to 1.0 (full volume, no filtering), 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

Requires the built in lowpass to be created with
```
FMOD_INIT_CHANNEL_LOWPASS
```
or
```
FMOD_INIT_CHANNEL_DISTANCEFILTER
```. 
See Also

- ChannelControl::getLowPassGain
Firelight Technologies FMOD Studio API
ChannelControl::setMixLevelsInput

Sets the incoming volume level for each channel of a multi-channel sound. This is a helper to avoid calling ChannelControl::setMixMatrix.

A multi-channel sound is a single sound that contains from 1 to 32 channels of sound data, in an interleaved fashion. If in the extreme case, a 32ch wave file was used, an array of 32 floating point numbers denoting their volume levels would be passed in to the levels parameter, and 32 for the numlevels parameter.

C++ Syntax

FMOD_RESULT ChannelControl::setMixLevelsInput(
    float *levels,
    int numlevels
);
Parameters

*levels*

Array of volume levels for each incoming channel.

*numlevels*

Number of levels in the array, from 0 to 32 inclusive.
Return Values

If the function succeeds then the return value 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 example use case for this function is if the sound file has multiple channels in it with different musical parts to it, but they are all in sync with each other. This function can be used to fade in and out different tracks of the sound or to solo/mute tracks within it.

NOTE: This function overwrites any pan/output mixlevel by overwriting the ChannelControl's matrix if it exists. It will create an NxN matrix where the output levels are the same as the input levels. If you wish to fold this down to a lower channel count mix rather than staying at the input channel count, either create a custom matrix instead and use ChannelControl::setMixMatrix, or add a new DSP after the fader, that has a different channel format (ie with ChannelControl::getDSP and DSP::setChannelFormat).

NOTE: Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.
See Also

- ChannelControl::setMixMatrix
- ChannelControl::getMixMatrix
- ChannelControl::getDSP
- DSP::setChannelFormat
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
Sets the speaker volume levels for each speaker individually, this is a helper to avoid calling ChannelControl::setMixMatrix.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setMixLevelsOutput(
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float surroundleft,
    float surroundright,
    float backleft,
    float backright
);
```
**Parameters**

*frontleft*

Volume level for the front left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

*frontright*

Volume level for the front right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

*center*

Volume level for the center speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

*lfe*

Volume level for the subwoofer speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

*surroundleft*

Volume level for the surround left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

*surroundright*

Volume level for the surround right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

*backleft*

Volume level for the back left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

*backright*
Volume level for the back right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (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

NOTE: This function overwrites any pan/mixlevel by overwriting the ChannelControl's Matrix.

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion. 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 ChannelControl::setMixMatrix.
See Also

- `ChannelControl::setMixMatrix`
- `ChannelControl::getMixMatrix`
Firelight Technologies FMOD Studio API
ChannelControl::setMixMatrix

Sets a 2D pan matrix that maps input channels (columns) to output speakers (rows).

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setMixMatrix(
    float *matrix,
    int outchannels,
    int inchannels,
    int matrixhop
);
```
Parameters

matrix

Address of a 2 dimensional array of volume levels in row-major order. Each row represents an output speaker, each column represents an input channel.

outchannels

Number of output channels (rows) in the matrix being passed in, from 0 to FMOD_MAX_CHANNEL_WIDTH inclusive.

inchannels

Number of input channels (columns) in the matrix being passed in, from 0 to FMOD_MAX_CHANNEL_WIDTH inclusive.

matrixhop

The width (total number of columns) of the matrix. Optional. If this is 0, inchannels will be taken as the width of the matrix. Maximum of FMOD_MAX_CHANNEL_WIDTH.
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

The gain for input channel 's' to output channel 't' is matrix[t * matrixhop + s].

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

The matrix size will generally be the size of the number of channels in the current speaker mode. Use `System::getSoftwareFormat` to determine this.

If a matrix already exists then the matrix passed in will applied over the top of it. The input matrix can be smaller than the existing matrix.

A 'unit' matrix allows a signal to pass through unchanged. For example for a 5.1 matrix a unit matrix would look like this:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
See Also

- `ChannelControl::getMixMatrix`
- `ChannelControl::setPan`
- `ChannelControl::setMixLevelsOutput`
- `ChannelControl::setMixLevelsInput`
- `System::getSoftwareFormat`
- `FMOD_MAX_CHANNEL_WIDTH`
Firelight Technologies FMOD Studio API
ChannelControl::setMode

Changes some attributes for a channel or channelgroup based on the mode passed in.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setMode(
    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`
- `FMOD_2D`
- `FMOD_3D`
- `FMOD_3D_HEADRELATIVE`
- `FMOD_3D_WORLDRELATIVE`
- `FMOD_3D_INVERSEROLLOFF`
- `FMOD_3D_LINEARROLLOFF`
- `FMOD_3D_LINEARSQUAREROLLOFF`
- `FMOD_3D_CUSTOMROLLOFF`
- `FMOD_3D_IGNOREGEOMETRY`
- `FMOD_VIRTUAL_PLAYFROMSTART`

Issues with streamed audio:

When changing the loop mode, sounds created with `System::createStream` or `FMOD_CREATESTREAM` may have already 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 by calling `Channel::setPosition`. Note this will usually only happen if you have sounds or loop points that are smaller than the stream decode buffer size.

Issues with PCM samples:

When changing the loop mode of sounds created with with `System::createSound` or `FMOD_CREATESAMPLE`, 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 (i.e. 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` or `FMOD_VIRTUAL_PLAYFROMSTART` is not specified, the flag will be cleared if it was specified previously.
See Also

- `Channel::getMode`
- `Channel::setPosition`
- `Sound::setMode`
- `System::createStream`
- `System::createSound`
- `System::setStreamBufferSize`
- `FMOD_CREATE_SOUND_INFO`
- `FMOD_MODE`
Firelight Technologies FMOD Studio API
ChannelControl::setMute

Sets the mute state effectively silencing it or returning it to its normal volume.

C++ Syntax

```
FMOD_RESULT ChannelControl::setMute(
    bool mute
);
```
Parameters

mute

Mute state, true = mute (silent), false = 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

Each channel and channel group has its own mute state, muting a channel group will mute all child channels but will not affect their individual setting. Calling `ChannelControl::getMute` will always return the value you set.
See Also

- ChannelControl::getMute
Firelight Technologies FMOD Studio API
ChannelControl::setPan

Sets the pan level, this is a helper to avoid calling ChannelControl::setMixMatrix.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setPan(
    float pan
);
```
Parameters

*pan*

Pan level, from -1.0 (left) to 1.0 (right), default = 0 (center).
**Return Values**

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

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

- ChannelControl::setMixMatrix
- ChannelControl::getMixMatrix
Firelight Technologies FMOD Studio API
ChannelControl::setPaused

Sets the paused state.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setPaused(
    bool paused
);
```
Parameters

paused

Paused state, 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

Each channel and channel group has its own paused state, pausing a channel group will pause all contained channels but will not affect their individual setting. Calling `ChannelControl::getPaused` will always return the value you set.
See Also

- [ChannelControl::getPaused](#)

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
**ChannelControl::setPitch**

Sets the pitch value.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelControl::setPitch(
    float pitch
);
```
Parameters

pitch

Pitch value, 0.5 = half pitch, 2.0 = double pitch, etc 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 scales existing frequency values by the pitch.
See Also

- ChannelControl::getPitch
- Channel::setFrequency
- Channel::getFrequency
Firelight Technologies FMOD Studio API
ChannelControl::setReverbProperties

Sets the wet level (or send level) of a particular reverb instance.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setReverbProperties(
    int instance,
    float wet
);```
Parameters

instance

Index of the particular reverb instance to target, from 0 to `FMOD_REVERB_MAXINSTANCES` inclusive.

wet

Send level for the signal to the reverb, from 0 (none) to 1.0 (full), default = 1.0 for Channels, 0.0 for ChannelGroups. 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

A Channel is automatically connected to all existing reverb instances due to the default wet level of 1.0. A ChannelGroup however will not send to any reverb by default requiring an explicit call to this function.

A ChannelGroup reverb is optimal for the case where you want to send 1 mixed signal to the reverb, rather than a lot of individual channel reverb sends. It is advisable to do this to reduce CPU if you have many Channels inside a ChannelGroup.

Keep in mind when setting a wet level for a ChannelGroup, any Channels under that ChannelGroup will still have their existing sends to the reverb. To avoid this doubling up you should explicitly set the Channel wet levels to 0.0.
See Also

- `ChannelControl::getReverbProperties`
Firelight Technologies FMOD Studio API
ChannelControl::setUserData

Sets a user value that can be retrieved with ChannelControl::getUserData.

C++ Syntax

FMOD_RESULT ChannelControl::setUserData(
   void *userdata
);

Parameters

*userdata*

Data that the user wishes stored within 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

You can use this to store a pointer to any wrapper or internal class that is associated with this object. This is especially useful for callbacks where you need to get back access to your own objects.
See Also

- ChannelControl::getUserData
Firelight Technologies FMOD Studio API
ChannelControl::setVolume

Sets the volume level linearly.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setVolume(
    float volume
);
```
Parameters

\textit{volume}

Linear volume level, 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

Volume level can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

Sound::setDefault can be used to change the default volume for any channels played using that sound.
See Also

- ChannelControl::getVolume
- Sound::setDefaults
Firelight Technologies FMOD Studio API
ChannelControl::setVolumeRamp

Sets whether the channel automatically ramps when setting volumes.

C++ Syntax

```cpp
FMOD_RESULT ChannelControl::setVolumeRamp(
    bool ramp
);
```
Parameters

*ramp*

Whether to enable volume ramping.
Return Values

If the function succeeds then the return value 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 changing volumes on a non-paused channel, FMOD normally adds a small ramp to avoid a pop sound. This function allows that setting to be overridden and volume changes to be applied immediately.
See Also

- ChannelControl::getVolumeRamp
Firelight Technologies FMOD Studio API
ChannelControl::stop

Stops the channel (or all channels in the channel group) from playing. Makes it available for re-use by the priority system.

C++ Syntax

FMOD_RESULT ChannelControl::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.
See Also

- System::playSound
Firelight Technologies FMOD Studio API
Channel
Functions

Channel::addDSP  Channel::addFadePoint  
Channel::get3DAttributes  
Channel::get3DConeOrientation  
Channel::get3DConeSettings  
Channel::get3DCustomRolloff  
Channel::get3DDistanceFilter  
Channel::get3DDopplerLevel  
Channel::get3DLevel  
Channel::get3DMinMaxDistance  
Channel::get3DOcclusion  
Channel::get3DSpread  
Channel::getAudibility  
Channel::getChannelGroup  
Channel::getCurrentSound  
Channel::getDSP  
Channel::getDSPClock  
Channel::getDSPIndex  
Channel::getDelay  
Channel::getFadePoints  
Channel::getFrequency  
Channel::getIndex  
Channel::getLoopCount  
Channel::getLoopPoints  
Channel::getLowPassGain  
Channel::getMixMatrix  
Channel::getMode  
Channel::getMute  
Channel::getNumDSPs  
Channel::getPaused  
Channel::getPitch  
Channel::getPosition  
Channel::getPriority  
Channel::getReverbProperties  
Channel::getSystemObject  
Channel::getUserData
Channel::getVolume
Channel::getVolumeRamp
Channel::isPlaying
Channel::isVirtual
Channel::removeDSP
Channel::removeFadePoints
Channel::set3DAttributes
Channel::set3DConeOrientation
Channel::set3DConeSettings
Channel::set3DCustomRolloff
Channel::set3DDistanceFilter
Channel::set3DDopplerLevel
Channel::set3DLevel
Channel::set3DMinMaxDistance
Channel::set3DOcclusion
Channel::set3DSpread
Channel::setCallback
Channel::setChannelGroup
Channel::setDSPIndex
Channel::setDelay
Channel::setFadePointRamp
Channel::setFrequency
Channel::setLoopCount
Channel::setLoopPoints
Channel::setLowPassGain
Channel::setMixLevelsInput
Channel::setMixLevelsOutput
Channel::setMixMatrix
Channel::setMode
Channel::setMute
Channel::setPan
Channel::setPaused
Channel::setPitch
Channel::setPosition
Channel::setPriority
Channel::setReverbProperties
Channel::setUserData
Channel::setVolume
Channel::setVolumeRamp
Firelight Technologies FMOD Studio API
Channel::addDSP

Add a pre-created DSP unit to the specified index in the DSP chain.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::addDSP(
    int index,
    DSP *dsp
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_AddDSP(
    FMOD_CHANNEL *channel,
    int index,
    FMOD_DSP *dsp
);
```

**C# Syntax**

```csharp
RESULT Channel.addDSP(
    int index,
    DSP dsp
);
```

**JavaScript Syntax**

```javascript
Channel.addDSP(
    index,
    dsp
);
```
**Parameters**

index  
Offset to add this DSP unit at in the DSP chain, see [FMOD_CHANNELCONTROL_DSP_INDEX](#) for special named offsets.

dsp  
Pointer to a pre-created DSP unit to be inserted at the specified offset.
Return Values

If the function succeeds then the return 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::createDSPByPlugin
- ChannelControl::removeDSP
- ChannelControl::getDSP
- ChannelControl::getNumDSPs
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
Channel::addFadePoint

Add a volume point to fade from or towards, using a clock offset and 0 to 1 volume level.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::addFadePoint(
    unsigned long long dspclock,
    float volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_AddFadePoint(
    FMOD_CHANNEL *channel,
    unsigned long long dspclock,
    float volume
);
```

**C# Syntax**

```csharp
RESULT Channel.addFadePoint(
    ulong dspclock,
    float volume
);
```

**JavaScript Syntax**

```javascript
Channel.addFadePoint(
    dspclock,
    volume
);
```
Parameters

dspclock
    DSP clock of the parent channel group to set the fade point volume.
volume
    Volume level where 0 is silent and 1.0 is normal volume. Amplification is supported.
Return Values

If the function succeeds then the return value 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 every fade point, FMOD will do a per sample volume ramp between them. It will scale with the current Channel or ChannelGroup's volume.

// Ramp from full volume to half volume over the next 4096 samples
FMOD_RESULT result;
unsigned long long parentclock;
result = target->getDSPClock(NULL, &parentclock);
result = target->addFadePoint(parentclock, 1.0f);
result = target->addFadePoint(parentclock + 4096, 0.5f);
See Also

- `ChannelControl::removeFadePoints`
- `ChannelControl::setFadePointRamp`
- `ChannelControl::getFadePoints`
Firelight Technologies FMOD Studio API
Channel::get3DAttributes

Retrieves the position and velocity used to apply panning, attenuation and doppler.

C++ Syntax

```cpp
FMOD_RESULT Channel::get3DAttributes(
    FMOD_VECTOR *pos,
    FMOD_VECTOR *vel,
    FMOD_VECTOR *alt_pan_pos
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Get3DAttributes(
    FMOD_CHANNEL *channel,
    FMOD_VECTOR *pos,
    FMOD_VECTOR *vel,
    FMOD_VECTOR *alt_pan_pos
);
```

C# Syntax

```csharp
RESULT Channel.get3DAttributes(
    out VECTOR pos,
    out VECTOR vel,
    out VECTOR alt_pan_pos
);
```

JavaScript Syntax

```javascript
Channel.get3DAttributes(
    pos, // writes value to pos.val
    vel, // writes value to vel.val
    alt_pan_pos // writes value to alt_pan_pos.val
);
```
Parameters

pos
Address of a variable that receives the position in 3D space used for panning and attenuation. Optional, specify 0 or NULL to ignore.

vel
Address of a variable that receives the velocity in 'distance units per second' (see remarks) in 3D space. Optional, specify 0 or NULL to ignore.

alt_pan_pos
(Unimplemented).
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

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

- `ChannelControl::set3DAttributes`
- `System::set3DSettings`
- `FMOD_VECTOR`
Firelight Technologies FMOD Studio API
Channel::get3DConeOrientation

Retrieves the orientation of the sound projection cone.

C++ Syntax

FMOD_RESULT Channel::get3DConeOrientation(
    FMOD_VECTOR *orientation
);

C Syntax

FMOD_RESULT FMOD_Channel_Get3DConeOrientation(
    FMOD_CHANNEL *channel,
    FMOD_VECTOR *orientation
);

C# Syntax

RESULT Channel.get3DConeOrientation(
    out VECTOR orientation
);

JavaScript Syntax

Channel.get3DConeOrientation( 
    orientation // writes value to orientation.va
);
Parameters

orientation

Address of a variable that receives the coordinates of the sound cone orientation vector, 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

- ChannelControl::set3DConeOrientation
Firelight Technologies FMOD Studio API
Channel::get3DConeSettings

Retrieves the angles that define the sound projection cone including the volume when outside the 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
);

C# Syntax

RESULT Channel.get3DConeSettings(
    out float insideconeangle,
    out float outsideconeangle,
    out float outsidevolume
);

JavaScript Syntax

Channel.get3DConeSettings(
    insideconeangle, // writes value to insideconeangle
    outsideconeangle, // writes value to outsideconeangle
    outsidevolume     // writes value to outsidevolume.
);
**Parameters**

insideconeangle
Address of a variable that receives the inside cone angle, in degrees. This is the angle within which the sound is at its normal volume.

outsideconeangle
Address of a variable that receives the outside cone angle, in degrees. This is the angle outside of which the sound is at its outside volume.

outsidevolume
Address of a variable that receives the cone outside 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

- ChannelControl::set3DConeSettings
Firelight Technologies FMOD Studio API
Channel::get3DCustomRolloff

Retrieves a pointer to the current custom rolloff curve.

C++ Syntax

FMOD_RESULT Channel::get3DCustomRolloff(
    FMOD_VECTOR **points,
    int *numpoints
);

C Syntax

FMOD_RESULT FMOD_Channel_Get3DCustomRolloff(
    FMOD_CHANNEL *channel,
    FMOD_VECTOR **points,
    int *numpoints
);

C# Syntax

RESULT Channel.get3DCustomRolloff(
    out IntPtr points,
    out int numpoints
);

JavaScript Syntax

Channel.get3DCustomRolloff(
    points, // writes value to points.val
    numpoints // writes value to numpoints.val
);
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

- ChannelControl::set3DCustomRolloff
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
Channel::get3DDistanceFilter

Retrieve the settings for the 3D distance filter properties for a Channel or Channel Group.

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
);
```

C# Syntax

```csharp
RESULT Channel.get3DDistanceFilter(
    out bool custom,
    out float customLevel,
    out float centerFreq
);
```

JavaScript Syntax

```javascript
Channel.get3DDistanceFilter(
    custom,       // writes value to custom.val
    customLevel,  // writes value to customLevel.val
    centerFreq    // writes value to centerFreq.val
);
```
Parameters

custom
   Address of a variable to receive the enabled/disabled state of the FMOD distance rolloff calculation. Default = false.

customLevel
   Address of a variable to receive the manual user attenuation, where 1.0 = no attenuation and 0 = complete attenuation. Default = 1.0.

centerFreq
   Address of a variable to receive center frequency in hz for the high-pass filter used to simulate distance attenuation, from 10.0 to 22050.0. Default = 1500.0.
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

- ChannelControl::set3DDistanceFilter
Firelight Technologies FMOD Studio API
Channel::get3DDopplerLevel

Retrieves the amount by which doppler is scaled.

C++ Syntax

FMOD_RESULT Channel::get3DDopplerLevel(
    float *level
);

C Syntax

FMOD_RESULT FMOD_Channel_Get3DDopplerLevel(
    FMOD_CHANNEL *channel,
    float *level
);

C# Syntax

RESULT Channel.get3DDopplerLevel(
    out float level
);

JavaScript Syntax

Channel.get3DDopplerLevel(  
    level  // writes value to level.val
);
Parameters

level

Address of a variable to receives the doppler scale from 0.0 (none), to 1.0 (normal) to 5.0 (exaggerated).
Return Values

If the function succeeds then the return 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

- [ChannelControl::set3DDopplerLevel](#)
Firelight Technologies FMOD Studio API
**Channel::get3DLevel**

Retrieves the current 3D mix level set by ChannelControl::set3DPanLevel.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::get3DLevel(
    float *level
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_Get3DLevel(
    FMOD_CHANNEL *channel,
    float *level
);
```

**C# Syntax**

```csharp
RESULT Channel.get3DLevel(
    out float level
);
```

**JavaScript Syntax**

```javascript
Channel.get3DLevel(    // writes value to level.val
    level
    // writes value to level.val
);
Parameters

level

3D pan level from 0.0 (attenuation is ignored and panning as set by 2D panning functions) to 1.0 (pan and attenuate according to 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

- ChannelControl::set3DPanLevel
Firelight Technologies FMOD Studio API
Channel::get3DMinMaxDistance

Retrieves the minimum and maximum audible distance.

C++ Syntax

FMOD_RESULT Channel::get3DMinMaxDistance(
  float *mindistance,
  float *maxdistance
);

C Syntax

FMOD_RESULT FMOD_Channel_Get3DMinMaxDistance(
  FMOD_CHANNEL *channel,
  float *mindistance,
  float *maxdistance
);

C# Syntax

RESULT Channel.get3DMinMaxDistance(
  out float mindistance,
  out float maxdistance
);

JavaScript Syntax

Channel.get3DMinMaxDistance(
  mindistance,              // writes value to mindistance.val
  maxdistance               // writes value to maxdistance.val
);
**Parameters**

mindistance

Address of a variable that receives the minimum volume distance in 'units' (see remarks). Optional, specify 0 or NULL to ignore.

maxdistance

Address of a variable that receives the maximum volume distance in 'units' (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

- ChannelControl::set3DMinMaxDistance
- System::set3DSettings

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
Channel::get3DOcclusion

Retrieves the occlusion factors.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::get3DOcclusion(
    float *directocclusion,
    float *reverbocclusion
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_Get3DOcclusion(
    FMOD_CHANNEL *channel,
    float *directocclusion,
    float *reverbocclusion
);
```

**C# Syntax**

```csharp
RESULT Channel.get3DOcclusion(
    out float directocclusion,
    out float reverbocclusion
);
```

**JavaScript Syntax**

```javascript
Channel.get3DOcclusion(
    directocclusion, // writes value to directocclusion
    reverbocclusion, // writes value to reverbocclusion
);
```
Parameters

directocclusion
Address of a variable that receives the occlusion factor for the direct path.
reverbocclusion
Address of a variable that receives the occlusion factor for the reverb mix.
Return Values

If the function succeeds then the return 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

- ChannelControl::set3DOcclusion
Firelight Technologies FMOD Studio API
Channel::get3DSpread

Retrieves the spread of a 3D sound in speaker space.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::get3DSpread(
    float *angle
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_Get3DSpread(
    FMOD_CHANNEL *channel,
    float *angle
);
```

**C# Syntax**

```csharp
RESULT Channel.get3DSpread(
    out float angle
);
```

**JavaScript Syntax**

```javascript
Channel.get3DSpread(
    angle
); // writes value to angle.val
```
Parameters

angle
   Address of a variable that receives the speaker spread angle.
Return Values

If the function succeeds then the return 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

- ChannelControl::set3DSpread
Firelight Technologies FMOD Studio API
Channel::getAudibility

Retrieves the combined volume after 3D spatialization and geometry occlusion calculations including any volumes set via the API.

C++ Syntax

```cpp
FMOD_RESULT Channel::getAudibility(
    float *audibility
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetAudibility(
    FMOD_CHANNEL *channel,
    float *audibility
);
```

C# Syntax

```csharp
RESULT Channel.getAudibility(
    out float audibility
);
```

JavaScript Syntax

```javascript
Channel.getAudibility(
    audibility // writes value to audibility.val
);
```
Parameters

audibility
   Address of a variable that receives the 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 result of all volume modifiers. This value is used by the virtual channel system to order its channels between real and virtual.

See the Virtual Voice System page for more details about how the audibility is calculated.
See Also

- Channel::isVirtual
- ChannelControl::getVolume
- ChannelControl::get3DOcclusion
- ChannelControl::get3DAttributes
- FMOD_DSP_PARAMETER_OVERALLGAIN
- FMOD_DSP_PARAMETER_DATA_TYPE_OVERALLGAIN
Firelight Technologies FMOD Studio API
Channel::getChannelGroup

Retrieves the currently assigned channel group for the channel.

**C++ Syntax**

```
FMOD_RESULT Channel::getChannelGroup(
    FMOD::ChannelGroup **channelgroup
);
```

**C Syntax**

```
FMOD_RESULT FMOD_Channel_GetChannelGroup(
    FMOD_CHANNEL *channel,
    FMOD_CHANNELGROUP **channelgroup
);
```

**C# Syntax**

```
RESULT Channel.getChannelGroup(
    out ChannelGroup channelgroup
);
```

**JavaScript Syntax**

```
Channel.getChannelGroup(
    channelgroup // writes value to channelgroup.v
);
```
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 Studio API
Channel::getCurrentSound

Retrieves the currently playing sound for this channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getCurrentSound(
    FMOD::Sound **sound
);
```

C Syntax

```c
FMODRESULT FMOD_Channel_GetCurrentSound(
    FMOD_CHANNEL *channel,
    FMOD_SOUND **sound
);
```

C# Syntax

```csharp
RESULT Channel.GetCurrentSound(
    out Sound sound
);
```

JavaScript Syntax

```javascript
Channel.getCurrentSound(
    sound // writes value to sound.val
);
```
Parameters

sound
    Address of a variable that receives a 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
Firelight Technologies FMOD Studio API
Channel::getDSP

Retrieve the DSP unit at the specified index.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getDSP(
    int index,
    DSP **dsp
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetDSP(
    FMOD_CHANNEL *channel,
    int index,
    FMOD_DSP **dsp
);
```

**C# Syntax**

```csharp
RESULT Channel.getDSP(
    int index,
    out DSP dsp
);
```

**JavaScript Syntax**

```javascript
Channel.getDSP(
    index,
    dsp // writes value to dsp.val
);
```
Parameters

index
   Offset into the DSP chain, see
   `FMOD_CHANNELCONTROL_DSP_INDEX` for special named offsets.

dsp
   Address of a variable to receive a pointer to the requested 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.
See Also

- `ChannelControl::addDSP`
- `ChannelControl::removeDSP`
- `ChannelControl::getNumDSPs`
- `FMOD_CHANNELCONTROL_DSP_INDEX`
Firelight Technologies FMOD Studio API
Channel::getDSPClock

Retrieves the DSP clock values which count up by the number of samples per second in the software mixer, i.e. if the default sample rate is 48KHz, the DSP clock increments by 48000 per second.

C++ Syntax

```cpp
FMOD_RESULT Channel::getDSPClock(
    unsigned long long *dspclock,
    unsigned long long *parentclock
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetDSPClock(
    FMOD_CHANNEL *channel,
    unsigned long long *dspclock,
    unsigned long long *parentclock
);
```

C# Syntax

```csharp
RESULT Channel.getDSPClock(
    out ulong dspclock,
    out ulong parentclock
);
```

JavaScript Syntax

```javascript
Channel.getDSPClock(
    dspclock, // writes value to dspclock.val
    parentclock // writes value to parentclock.val
);
```
Parameters

dspclock
   Address of a variable to receive the DSP clock value for the head DSP node.

parentclock
   Address of a variable to receive the DSP clock value for the tail DSP node.
Return Values

If the function succeeds then the return value 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 `ChannelControl::setDelay` to play a sound on an exact tick in the future, or stop it in the future.

Note that when delaying a channel or channel group you want to sync it to the parent channel group DSP clock value, not its own DSP clock value.
See Also

- ChannelControl::setDelay
- ChannelControl::getDelay
Firelight Technologies FMOD Studio API
**Channel::getDSPIndex**

Retrieves the index in the DSP chain of the provided DSP.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getDSPIndex(
    DSP *dsp,
    int *index
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetDSPIndex(
    FMOD_CHANNEL *channel,
    FMOD_DSP *dsp,
    int *index
);
```

**C# Syntax**

```csharp
RESULT Channel.getDSPIndex(
    DSP dsp,
    out int index
);
```

**JavaScript Syntax**

```javascript
Channel.getDSPIndex(
    dsp,
    index // writes value to index.val
);
```
Parameters

dsp
   Pointer to a DSP unit that exists in the DSP chain.
index
   Address of a variable to receive the offset in the DSP chain of the specified DSP.
Return Values

If the function succeeds then the return 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

- ChannelControl::setDSPIndex
Firelight Technologies FMOD Studio API
Channel::getDelay

Retrieves a start (and/or stop) time relative to the parent channel group DSP clock, with sample accuracy.

C++ Syntax

```cpp
FMOD_RESULT Channel::getDelay(
    unsigned long long *dspclock_start,
    unsigned long long *dspclock_end,
    bool *stopchannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetDelay(
    FMOD_CHANNEL *channel,
    unsigned long long *dspclock_start,
    unsigned long long *dspclock_end,
    FMOD_BOOL *stopchannels
);
```

C# Syntax

```csharp
RESULT Channel.getDelay(
    out ulong dspclock_start,
    out ulong dspclock_end,
    out bool stopchannels
);
```

JavaScript Syntax

```javascript
Channel.getDelay(
    dspclock_start,     // writes value to dspclock_start
    dspclock_end,       // writes value to dspclock_end.v
    stopchannels        // writes value to stopchannels.v
);
```
**Parameters**

dspclock_start
Address of a variable that receives the DSP clock of the parent channel group to audibly start playing sound at. Optional, specify 0 or NULL to ignore.

dspclock_end
Address of a variable that receives the DSP clock of the parent channel group to audibly stop playing sound at. Optional, specify 0 or NULL to ignore.

stopchannels
Address of a variable that receives TRUE = stop according to `ChannelControl::isPlaying`. FALSE = remain 'active' and a new start delay could start playback again at a later time. 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

- ChannelControl::setDelay
- ChannelControl::getDSPClock
- ChannelControl::isPlaying
Firelight Technologies FMOD Studio API
Channel::getFadePoints

Retrieve information about fade points stored within a Channel or ChannelGroup.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getFadePoints(
    unsigned int *numpoints,
    unsigned long long *point_dspclock,
    float *point_volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetFadePoints(
    FMOD_CHANNEL *channel,
    unsigned int *numpoints,
    unsigned long long *point_dspclock,
    float *point_volume
);
```

**C# Syntax**

```csharp
RESULT Channel.getFadePoints(
    ref uint numpoints,
    ulong[] point_dspclock,
    float[] point_volume
);
```

**JavaScript Syntax**

```javascript
Channel.getFadePoints(
    numpoints, // writes value to numpoints.val
    point_dspclock, // writes value to point_dspclock
    point_volume // writes value to point_volume.v
);
```
Parameters

numpoints
Address of a variable to receive the number of fade points stored within the Channel or ChannelGroup.

point_dspclock
Address of a variable to receive an array of 64bit clock values. Can be 0 or NULL.

point_volume
Address of a variable to receive an array of floating point volume values. Can be 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 **FMOD_RESULT** enumeration.
Remarks

To first get the number of points for memory purposes, and not store any data, call this function with point_dpsclock and point_volume parameters being 0 or NULL.
See Also

- `ChannelControl::addFadePoint`
- `ChannelControl::removeFadePoints`
Firelight Technologies FMOD Studio API
Channel::getFrequency

Retrieves the channel frequency or playback rate, in Hz.

**C++ Syntax**

```c++
FMOD_RESULT Channel::getFrequency(
    float *frequency
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetFrequency(
    FMOD_CHANNEL *channel,
    float *frequency
);
```

**C# Syntax**

```csharp
RESULT Channel.getFrequency(
    out float frequency
);
```

**JavaScript Syntax**

```javascript
Channel.getFrequency(
    frequency // writes value to frequency.val
);
```
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 Studio API
Channel::getIndex

Retrieves the internal channel index for a channel.

C++ Syntax

FMOD_RESULT Channel::getIndex(
    int *index
);

C Syntax

FMOD_RESULT FMOD_Channel_GetIndex(
    FMOD_CHANNEL *channel,
    int *index
);

C# Syntax

RESULT Channel.getIndex(
    out int index
);

JavaScript Syntax

Channel.getIndex(
    index // writes value to index.val
);
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.
See Also

- System::playSound
- System::init
Firelight Technologies FMOD Studio API
Channel::getLoopCount

Retrieves the current loop count for the specified channel.

C++ Syntax

```cpp
FMOD_RESULT Channel::getLoopCount(
    int *loopcount
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetLoopCount(
    FMOD_CHANNEL *channel,
    int *loopcount
);
```

C# Syntax

```csharp
RESULT Channel.getLoopCount(
    out int loopcount
);
```

JavaScript Syntax

```javascript
Channel.getLoopCount(      
    loopcount // writes value to loopcount.val
);
```
Parameters

loopcount
Address of a variable that receives the number of times to loop before stopping. 0 = oneshot, 1 = loop once then stop, -1 = loop forever.
Return Values

If the function succeeds then the return value 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 Studio API
Channel::getLoopPoints

Retrieves the loop points for the 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
);
```

**C# Syntax**

```csharp
RESULT Channel.getLoopPoints(
    out uint loopstart,
    TIMEUNIT loopstarttype,
    out uint loopend,
    TIMEUNIT loopendtype
);
```

**JavaScript Syntax**

```javascript
Channel.getLoopPoints(
    loopstart,          // writes value to loopstart.val
    loopstarttype,
    loopend,           // writes value to loopend.val
    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
   Time format used for the 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
   Time format used for the loop end point (see FMOD_TIMEUNIT).
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

- Channel::setLoopPoints
- FMOD_TIMEUNIT
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Channel.getLowPassGain(
    out float gain
);
```

**JavaScript Syntax**

```javascript
Channel.getLowPassGain(
    gain // writes value to gain.val
);
```
Parameters

gain
   Address of a variable that receives the linear gain level, from 0 (silent) to 1.0 (full volume).
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

- ChannelControl::setLowPassGain
Firelight Technologies FMOD Studio API
Channel::getMixMatrix

Retrieves a 2D pan matrix that maps input channels (columns) to output speakers (rows).

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getMixMatrix(
    float *matrix,
    int *outchannels,
    int *inchannels,
    int matrixhop
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetMixMatrix(
    FMOD_CHANNEL *channel,
    float *matrix,
    int *outchannels,
    int *inchannels,
    int matrixhop
);
```

**C# Syntax**

```csharp
RESULT Channel.getMixMatrix(
    float[] matrix,
    out int outchannels,
    out int inchannels,
    int inchannel_hop
);
```

**JavaScript Syntax**

```javascript
Channel.getMixMatrix(
    matrix, // writes value to matrix.val
    outchannels, // writes value to outchannels.val
    inchannels, // writes value to inchannels.val
    inchannel_hop
);
```
Parameters

matrix
Address of a 2 dimensional array of volume levels in row-major order. Each row represents an output speaker, each column represents an input channel.

outchannels
Address of a variable to receive the number of output channels (rows) in the matrix being passed in.

inchannels
Address of a variable to receive the number of input channels (columns) in the matrix being passed in.

matrixhop
The width (total number of columns) of the matrix. Optional. If this is 0, inchannels will be taken as the width of the matrix. Maximum of FMOD_MAX_CHANNEL_WIDTH.
Return Values

If the function succeeds then the return value 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 gain for input channel 's' to output channel 't' is matrix[t * matrixhop + s].

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

The matrix size will generally be the size of the number of channels in the current speaker mode. Use System::getSoftwareFormat to determine this.

Passing NULL for 'matrix' will allow you to query 'outchannels' and 'inchannels' without copying any data.
See Also

- ChannelControl::setMixMatrix
- System::getSoftwareFormat
- FMOD_MAX_CHANNEL_WIDTH
Firelight Technologies FMOD Studio API
Channel::getMode

Retrieves the mode bit flags for the 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
);
```

C# Syntax

```csharp
RESULT Channel.getMode(
    out MODE mode
);
```

JavaScript Syntax

```javascript
Channel.getMode( 
    mode // writes value to mode.val 
); 
```
Parameters

mode
   Address of a variable to receive the mode bits.
Return Values

If the function succeeds then the return 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
- FMOD_MODE
Firelight Technologies FMOD Studio API
Channel::getMute

Retrieves the mute state.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getMute(
    bool *mute
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetMute(
    FMOD_CHANNEL *channel,
    FMOD_BOOL *mute
);
```

**C# Syntax**

```csharp
RESULT Channel.getMute(
    out bool mute
);
```

**JavaScript Syntax**

```javascript
Channel.getMute(
    mute
) // writes value to mute.val
```
Parameters

mute
Address of a variable that receives the current mute state, true = mute (silent), false = 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

- [ChannelControl::setMute](#)
Firelight Technologies FMOD Studio API
Channel::getNumDSPs

Retrieves the number of DSP units in the DSP chain.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getNumDSPs(
    int *numdsps
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetNumDSPs(
    FMOD_CHANNEL *channel,
    int *numdsps
);
```

**C# Syntax**

```csharp
RESULT Channel.getNumDSPs(
    out int numdsps
);
```

**JavaScript Syntax**

```javascript
Channel.getNumDSPs(
    numdsps
    // writes value to numdsps.val
);
```
Parameters

numdsps
Address of a variable that receives the number of DSP units in the DSP chain.
Return Values

If the function succeeds then the return 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

- `ChannelControl::addDSP`
- `ChannelControl::removeDSP`
- `ChannelControl::getDSP`
Firelight Technologies FMOD Studio API
Channel::getPaused

Retrieves the paused state.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getPaused(
    bool *paused
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetPaused(
    FMOD_CHANNEL *channel,
    FMOD_BOOL *paused
);
```

**C# Syntax**

```csharp
RESULT Channel.getPaused(
    out bool paused
);
```

**JavaScript Syntax**

```javascript
Channel.getPaused(
    paused // writes value to paused.val
);
```
Parameters

paused
Address of a variable that receives the current paused state, true = paused, false = not paused.
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

- `ChannelControl::setPaused`
Firelight Technologies FMOD Studio API
Channel::getPitch

Retrieves the pitch value.

C++ Syntax

```cpp
FMOD_RESULT Channel::getPitch(
    float *pitch
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetPitch(
    FMOD_CHANNEL *channel,
    float *pitch
);
```

C# Syntax

```csharp
RESULT Channel.getPitch(
    out float pitch
);
```

JavaScript Syntax

```javascript
Channel.getPitch(
    pitch
); // writes value to pitch.val
```
Parameters

pitch
Address of a variable to receive the pitch value, 0.5 = half pitch, 2.0 = double pitch, etc.
Return Values

If the function succeeds then the return 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

- `ChannelControl::setPitch`
- `Channel::setFrequency`
- `Channel::getFrequency`
Firelight Technologies FMOD Studio API
Channel::getPosition

Returns the current 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_Channel_GetPosition(
    FMOD_CHANNEL *channel,
    unsigned int *position,
    FMOD_TIMEUNIT postype
);
```

C# Syntax

```csharp
RESULT Channel.getPosition(
    out uint position,
    TIMEUNIT postype
);
```

JavaScript Syntax

```javascript
Channel.getPosition(
    position, // writes value to position.val
    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 **FMOD_TIMEUNIT_MODORDER** will not work with an MP3 file.
See Also

- `Channel::setPosition`
- `FMOD_TIMEUNIT`
Firelight Technologies FMOD Studio API
Channel::getPriority

Retrieves the priority for the channel.

C++ Syntax

FMOD_RESULT Channel::getPriority(
    int *priority
);

C Syntax

FMOD_RESULT FMOD_Channel_GetPriority(
    FMOD_CHANNEL *channel,
    int *priority
);

C# Syntax

RESULT Channel.getPriority(
    out int priority
);

JavaScript Syntax

Channel.getPriority(
    priority // writes value to priority.val
);
Parameters

priority
   Address of a variable that receives the priority for the channel, from 0 (most important) to 256 (least important).
Return Values

If the function succeeds then the return 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 Studio API
Channel::getReverbProperties

Retrieves the wet level (or send level) for a particular reverb instance.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getReverbProperties(
    int instance,
    float *wet
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetReverbProperties(
    FMOD_CHANNEL *channel,
    int instance,
    float *wet
);
```

**C# Syntax**

```csharp
RESULT Channel.getReverbProperties(
    int instance,
    out float wet
);
```

**JavaScript Syntax**

```javascript
Channel.getReverbProperties(
    instance,
    wet // writes value to wet.val
);
```
Parameters

instance
   Index of the particular reverb instance to target, from 0 to
   `FMOD_REVERB_MAXINSTANCES` inclusive.

wet
   Address of a variable that receives the send level for the signal to the
   reverb, from 0 (none) to 1.0 (full).
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

- ChannelControl::setReverbProperties
Firelight Technologies FMOD Studio API
Channel::getSystemObject

Retrieves the parent System object that created the channel or channel group.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getSystemObject(
    System **system
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetSystemObject(
    FMOD_CHANNEL *channel,
    FMOD_SYSTEM **system
);
```

**C# Syntax**

```csharp
RESULT Channel.getSystemObject(
    out System system
);
```

**JavaScript Syntax**

```javascript
Channel.getSystemObject(
    system // writes value to system.val
);
```
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
- System::playSound
Firelight Technologies FMOD Studio API
Channel::getUserData

Retrieves a user value that can be set with ChannelControl::setUserData.

C++ Syntax

```cpp
FMOD_RESULT Channel::getUserData(
    void **userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetUserData(
    FMOD_CHANNEL *channel,
    void **userdata
);
```

C# Syntax

```csharp
RESULT Channel.getUserData(
    out IntPtr userdata
);
```

JavaScript Syntax

```javascript
Channel.getUserData(
    userdata // writes value to userdata.val
);
```
Parameters

userdata
    Address of a variable to receive data that the user has stored 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.
See Also

- `ChannelControl::setUserData`
Firelight Technologies FMOD Studio API
**Channel::getVolume**

Retrieves the volume level.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::getVolume(
    float *volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_GetVolume(
    FMOD_CHANNEL *channel,
    float *volume
);
```

**C# Syntax**

```csharp
RESULT Channel.getVolume(
    out float volume
);
```

**JavaScript Syntax**

```javascript
Channel.getVolume(
    volume // writes value to volume.val
);
```
Parameters

volume
   Address of a variable to receive the linear volume 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.
See Also

- [ChannelControl::setVolume](#)
Firelight Technologies FMOD Studio API
Channel::getVolumeRamp

Retrieves whether volume ramp is enabled.

C++ Syntax

```cpp
FMOD_RESULT Channel::getVolumeRamp(
    bool *ramp
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_GetVolumeRamp(
    FMOD_CHANNEL *channel,
    FMOD_BOOL *ramp
);
```

C# Syntax

```csharp
RESULT Channel.getVolumeRamp(
    out bool ramp
);
```

JavaScript Syntax

```javascript
Channel.getVolumeRamp(  
    ramp                                      // writes value to ramp.val
);  
```
Parameters

ramp
   Address of a variable to receive the volume ramp 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

- ChannelControl::setVolumeRamp
Firelight Technologies FMOD Studio API
Channel::isPlaying

Retrieves the playing state.

C++ Syntax

```cpp
FMOD_RESULT Channel::isPlaying(
    bool *isplaying
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_IsPlaying(
    FMOD_CHANNEL  *channel,
    FMOD_BOOL  *isplaying
);
```

C# Syntax

```csharp
RESULT Channel.isPlaying(
    out bool isplaying
);
```

JavaScript Syntax

```javascript
Channel.isPlaying(
    isplaying // writes value to isplaying.val
);
```
Parameters

isplaying
  Address of a variable that receives the current playing status, true = currently playing a sound, false = 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 Studio API
Channel::isVirtual

Retrieves whether the channel is virtual (emulated) or not due to the 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
);
```

**C# Syntax**

```csharp
RESULT Channel.isVirtual(
    out bool isvirtual
);
```

**JavaScript Syntax**

```javascript
Channel.isVirtual(
    isvirtual // writes value to isvirtual.val
);
```
**Parameters**

**isvirtual**
Address of a variable that receives the virtual status. TRUE = inaudible and currently being emulated at no CPU cost, FALSE = real voice that 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

See the Virtual Voice System page for more details about how channel virtualization works.
See Also

- System::playSound
- ChannelControl::getAudibility
Channel::removeDSP

Remove a particular DSP unit from the DSP chain.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::removeDSP(
    DSP *dsp
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_RemoveDSP(
    FMOD_CHANNEL *channel,
    FMOD_DSP *dsp
);
```

**C# Syntax**

```csharp
RESULT Channel.removeDSP(
    DSP dsp
);
```

**JavaScript Syntax**

```javascript
Channel.removeDSP(
    dsp
);
```
Parameters

dsp
    Pointer to a DSP unit (that exists in the DSP chain) you wish to remove.
Return Values

If the function succeeds then the return 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

- ChannelControl::addDSP
- ChannelControl::getDSP
- ChannelControl::getNumDSPs
Firelight Technologies FMOD Studio API
Channel::removeFadePoints

Remove volume fade points on the timeline. This function will remove multiple fade points with a single call if the points lay between the 2 specified clock values (inclusive).

C++ Syntax

```cpp
FMOD_RESULT Channel::removeFadePoints(
    unsigned long long dspclock_start,
    unsigned long long dspclock_end
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_RemoveFadePoints(
    FMOD_CHANNEL *channel,
    unsigned long long dspclock_start,
    unsigned long long dspclock_end
);
```

C# Syntax

```csharp
RESULT Channel.removeFadePoints(
    ulong dspclock_start,
    ulong dspclock_end
);
```

JavaScript Syntax

```javascript
Channel.removeFadePoints(
    dspclock_start,
    dspclock_end
);
```
Parameters

dspclock_start
  DSP clock of the parent channel group to start removing fade points from.
dspclock_end
  DSP clock of the parent channel group to start removing fade points 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.
See Also

- ChannelControl::addFadePoint
- ChannelControl::getFadePoints
Firelight Technologies FMOD Studio API
Channel::set3DAttributes

Sets the position and velocity used to apply panning, attenuation and doppler.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DAttributes(
    const FMOD_VECTOR *pos,
    const FMOD_VECTOR *vel,
    const FMOD_VECTOR *alt_pan_pos
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DAttributes(
    FMOD_CHANNEL *channel,
    const FMOD_VECTOR *pos,
    const FMOD_VECTOR *vel,
    const FMOD_VECTOR *alt_pan_pos
);
```

C# Syntax

```csharp
RESULT Channel.set3DAttributes(
    ref VECTOR pos,
    ref VECTOR vel,
    ref VECTOR alt_pan_pos
);
```

JavaScript Syntax

```javascript
Channel.set3DAttributes(
    pos,
    vel,
    alt_pan_pos
);
```
Parameters

pos
   Position in 3D space used for panning and attenuation. Optional, specify 0 or NULL to ignore.

vel
   Velocity in 'distance units per second' (see remarks) in 3D space. Optional, specify 0 or NULL to ignore.

alt_pan_pos
   (Unimplemented).
Return Values

If the function succeeds then the return value 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 `ChannelControl::set3DSpread`.

Vectors should use your chosen coordinate system, see [3D sounds](#) for more information.
See Also

- ChannelControl::get3DAttributes
- ChannelControl::set3DSpread
- System::set3DSettings
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT Channel.set3DConeOrientation(
    ref VECTOR orientation
);
```

JavaScript Syntax

```javascript
Channel.set3DConeOrientation(orientation);
```
Parameters

orientation
    Coordinates of the sound cone orientation vector, 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.
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

- `ChannelControl::get3DConeOrientation`
- `ChannelControl::set3DConeSettings`
- `Sound::set3DConeSettings`
- `FMOD_VECTOR`
Firelight Technologies FMOD Studio API
Channel::set3DConeSettings

Sets the angles that define the sound projection cone including the volume when outside the 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
);
```

C# Syntax

```csharp
RESULT Channel.set3DConeSettings(
    float insideconeangle,
    float outsideconeangle,
    float outsidevolume
);
```

JavaScript Syntax

```javascript
Channel.set3DConeSettings(
    insideconeangle,
    outsideconeangle,
    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 0.0 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

To define the parameters per sound use Sound::set3DConeSettings.
See Also

- ChannelControl::get3DConeSettings
- ChannelControl::set3DConeOrientation
- Sound::set3DConeSettings
Firelight Technologies FMOD Studio API
Channel::set3DCustomRolloff

Sets a custom rolloff curve to define how audio will attenuate over distance. 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
);
```

C# Syntax

```csharp
RESULT Channel.set3DCustomRolloff(
    ref VECTOR points,
    int numpoints
);
```

JavaScript Syntax

```javascript
Channel.set3DCustomRolloff(
    points,
    numpoints
);
```
Parameters

points
  Array of FMOD_VECTOR structures where x = distance and y = volume from 0.0 to 1.0. z should be set to 0.

numpoints
  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 \texttt{FMOD\_3D\_CUSTOMROLLOFF} is set and the rolloff curve is 0, FMOD will revert to inverse curve rolloff.

Values set with ChannelControl::setMinMaxDistance are meaningless when \texttt{FMOD\_3D\_CUSTOMROLLOFF} is used, their values are ignored.

Here is an example of a custom array of points.

```c
static \texttt{FMOD\_VECTOR} curve[3] =
{
    { 0.0f, 1.0f, 0.0f },
    { 2.0f, 0.2f, 0.0f },
    { 20.0f, 0.0f, 0.0f }
};
```

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.

To define the parameters per sound use \texttt{Sound::set3DCustomRolloff}. 
See Also

- ChannelControl::get3DCustomRolloff
- ChannelControl::setMinMaxDistance
- Sound::set3DCustomRolloff
- FMOD_VECTOR
- FMOD_3D_CUSTOMROLLOFF
Firelight Technologies FMOD Studio API
Channel::set3DDistanceFilter

Control the behaviour of a 3D distance filter, whether to enable or disable it, and frequency characteristics.

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
);
```

C# Syntax

```csharp
RESULT Channel.set3DDistanceFilter(
    bool custom,
    float customLevel,
    float centerFreq
);
```

JavaScript Syntax

```javascript
Channel.set3DDistanceFilter(
    custom,
    customLevel,
    centerFreq
);
```
Parameters

custom
   Specify true to disable FMOD distance rolloff calculation. Default = false.
customLevel
   Specify a attenuation factor manually here, where 1.0 = no attenuation and 0 = complete attenuation. Default = 1.0.
centerFreq
   Specify a center frequency in hz for the high-pass filter used to simulate distance attenuation, from 10.0 to 22050.0. Default = 1500.0.
**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

- [ChannelControl::get3DDistanceFilter](#)
Firelight Technologies FMOD Studio API
Channel::set3DDopplerLevel

Sets the amount by which doppler is scaled.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::set3DDopplerLevel(
    float level
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_Set3DDopplerLevel(
    FMOD_CHANNEL *channel, 
    float level
);
```

**C# Syntax**

```csharp
RESULT Channel.set3DDopplerLevel(
    float level
);
```

**JavaScript Syntax**

```javascript
Channel.set3DDopplerLevel(
    level
);
```
Parameters

level

Doppler scale from 0.0 (none), to 1.0 (normal) to 5.0 (exaggerated), 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

- `ChannelControl::get3DDopplerLevel`
Firelight Technologies FMOD Studio API
Channel::set3DLevel

Sets how much the 3D engine has an effect on the channel, versus that set by 2D panning functions.

C++ Syntax

FMOD_RESULT Channel::set3DLevel(
    float level
);

C Syntax

FMOD_RESULT FMOD_Channel_Set3DLevel(
    FMOD_CHANNEL *channel,
    float level
);

C# Syntax

RESULT Channel.set3DLevel(
    float level
);

JavaScript Syntax

Channel.set3DLevel(
    level
);
Parameters

level

3D pan level from 0.0 (attenuation is ignored and panning as set by 2D panning functions) to 1.0 (pan and attenuate according to 3D position), 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

Only affects sounds created `FMOD_3D`.

2D panning functions include `ChannelControl::setPan`, `ChannelControl::setMixLevelsOutput`, `ChannelControl::setMixLevelsInput`, `ChannelControl::setMixMatrix`, etc.

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).
See Also

- ChannelControl::get3DLevel
- ChannelControl::setPan
- ChannelControl::setMixLevelsOutput
- ChannelControl::setMixLevelsInput
- ChannelControl::setMixMatrix
Firelight Technologies FMOD Studio API
Channel::set3DMinMaxDistance

Sets the minimum and maximum audible distance.

C++ Syntax

```
FMOD_RESULT Channel::set3DMinMaxDistance(
    float mindistance,
    float maxdistance
);
```

C Syntax

```
FMOD_RESULT FMOD_Channel_Set3DMinMaxDistance(
    FMOD_CHANNEL *channel,
    float mindistance,
    float maxdistance
);
```

C# Syntax

```
RESULT Channel.set3DMinMaxDistance(
    float mindistance,
    float maxdistance
);
```

JavaScript Syntax

```
Channel.set3DMinMaxDistance(
    mindistance,
    maxdistance
);
```
Parameters

mindistance
Minimum volume distance in 'units' (see remarks), default = 1.0.

maxdistance
Maximum volume distance in 'units' (see remarks), default = 10000.0.
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 the listener is in-between the minimum distance and the sound source the volume will be at its maximum. As the listener moves from the minimum distance to the maximum distance the sound will attenuate following the rolloff curve set. When outside the maximum distance the sound will no longer attenuate.

Minimum distance 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 small mindistance such as 0.1. This would cause it to attenuate quickly and dissapear 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 causing the volume to stay at its loudest until the listener was 100 meters away, then it would be hundreds of meters more before it would fade out.

Maximum distance 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 zero, 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.

To define the min and max distance per sound use Sound::set3DMinMaxDistance.

If FMOD_3D_CUSTOMROLLOFF is used, then these values are stored, but ignored in 3D processing.
See Also

- ChannelControl::get3DMinMaxDistance
- System::set3DSettings
- Sound::set3DMinMaxDistance
- FMOD_3D_CUSTOMROLLOFF
Firelight Technologies FMOD Studio API
**Channel::set3DOcclusion**

Sets the occlusion factors manually for when the FMOD geometry engine is not being used.

**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
);
```

**C# Syntax**

```csharp
RESULT Channel.set3DOcclusion(
    float directocclusion,
    float reverbocclusion
);
```

**JavaScript Syntax**

```javascript
Channel.set3DOcclusion(
    directocclusion,
    reverbocclusion
);
```
Parameters

directocclusion
   Occlusion factor for the direct path, from 0.0 (not occluded) to 1.0 (fully occluded), default = 0.0.
reverbocclusion
   Occlusion factor for the reverb mix, from 0.0 (not occluded) to 1.0 (fully occluded), default = 0.0.
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

Normally the volume is simply attenuated by the 'directocclusion' factor however if `FMOD_INIT_CHANNEL_LOWPASS` is specified frequency filtering will be used with a very small CPU hit.
See Also

- ChannelControl::get3DOcclusion
- FMOD_INIT_CHANNEL_LOWPASS
Firelight Technologies FMOD Studio API
Channel::set3DSpread

Sets the spread of a 3D sound in speaker space.

C++ Syntax

```cpp
FMOD_RESULT Channel::set3DSpread(
    float angle
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_Set3DSpread(
    FMOD_CHANNEL *channel,
    float angle
);
```

C# Syntax

```csharp
RESULT Channel.set3DSpread(
    float angle
);
```

JavaScript Syntax

```javascript
Channel.set3DSpread(
    angle
);
```
Parameters

angle
Speaker spread angle. 0 = all sound channels are located at the same speaker location and is 'mono'. 360 = all sound channels 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

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'.

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 channel located 15 degrees apart from each other in the speaker array.

Mono sounds are spread as if they were a stereo signal, i.e. the signal is split into 2. The power will remain the same as it spreads 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!
See Also

- `ChannelControl::get3DSpread`
Firelight Technologies FMOD Studio API
Channel::setCallback

Sets a callback to perform action for a specific event.

C++ Syntax

FMOD_RESULT Channel::setCallback(
    FMOD_CHANNELCONTROL_CALLBACK callback
);

C Syntax

FMOD_RESULT FMOD_Channel_SetCallback(
    FMOD_CHANNEL *channel,
    FMOD_CHANNELCONTROL_CALLBACK callback
);

C# Syntax

RESULT Channel.setCallback(
    CHANNEL_CALLBACK callback
);

JavaScript Syntax

Channel.setCallback(
    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 driven by `System::update` and will only occur when this function is called. This has the main advantage of far less complication due to thread issues, and allows all FMOD commands, including loading sounds and playing new sounds from the callback. It also allows any type of sound to have an end callback, no matter what it is. The only disadvantage is that 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` in between your return type and function name.

```cpp
FMOD_RESULT F_CALLBACK mycallback(FMOD_CHANNELCONTROL *chanControl,
{
    if (controlType == FMOD_CHANNELCONTROL_TYPE_CHANNEL)
    {
        FMOD::Channel *channel = (FMOD::Channel *)chanControl;
        // Channel specific functions here...
    }
    else
    {
        FMOD::ChannelGroup *group = (FMOD::ChannelGroup *)chanControl;
        // ChannelGroup specific functions here...
    }
    // ChannelControl generic functions here...
    return FMOD_OK;
}
```
See Also

- System::update
- FMOD_CHANNELCONTROL_CALLBACK
- FMOD_CHANNELCONTROL_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
Channel::setChannelGroup

Sets a channel to belong to a specified channel group. A channel group can contain many channels.

C++ Syntax

FMOD_RESULT Channel::setChannelGroup(
    FMOD::ChannelGroup *channelgroup
);

C Syntax

FMOD_RESULT FMOD_Channel_SetChannelGroup(
    FMOD_CHANNEL *channel,
    FMOD_CHANNELGROUP *channelgroup
);

C# Syntax

RESULT Channel.setChannelGroup(
    ChannelGroup channelgroup
);

JavaScript Syntax

Channel.setChannelGroup(
    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 Studio API
Channel::setDSPIndex

Moves the position in the DSP chain of a specified DSP unit.

C++ Syntax

```c++
FMOD_RESULT Channel::setDSPIndex(
    DSP *dsp,
    int index
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetDSPIndex(
    FMOD_CHANNEL *channel,
    FMOD_DSP *dsp,
    int index
);
```

C# Syntax

```csharp
RESULT Channel.setDSPIndex(
    DSP dsp,
    int index
);
```

JavaScript Syntax

```javascript
Channel.setDSPIndex(
    dsp,
    index
);
```
Parameters

dsp
   Pointer to a DSP unit that exists in the DSP chain.

index
   Offset in the DSP chain to move the DSP to, see
   FMOD_CHANNELCONTROL_DSP_INDEX for special named offsets.
Return Values

If the function succeeds then the return value 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 reordering DSP units inside a Channel or ChannelGroup so that processing can happen in the desired order.

You can verify the order of the DSP chain using iteration via `ChannelControl::getNumDSPs` and `ChannelControl::getDSP` or with the FMOD Profiler tool.
See Also

- ChannelControl::getDSPIndex
- ChannelControl::getNumDSPs
- ChannelControl::getDSP
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
Channel::setDelay

Sets a start (and/or stop) time relative to the parent channel group DSP clock, with sample accuracy.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setDelay(
    unsigned long long dspclock_start,
    unsigned long long dspclock_end,
    bool stopchannels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetDelay(
    FMOD_CHANNEL *channel,
    unsigned long long dspclock_start,
    unsigned long long dspclock_end,
    FMOD_BOOL stopchannels
);
```

**C# Syntax**

```csharp
RESULT Channel.setDelay(
    ulong dspclock_start,
    ulong dspclock_end,
    bool stopchannels
);
```

**JavaScript Syntax**

```javascript
Channel.setDelay(
    dspclock_start,
    dspclock_end,
    stopchannels
);
```
Parameters

dspclock_start
DSP clock of the parent channel group to audibly start playing sound at, a value of 0 indicates no delay.

dspclock_end
DSP clock of the parent channel group to audibly stop playing sound at, a value of 0 indicates no delay.

stopchannels
TRUE = stop according to ChannelControl::isPlaying. FALSE = remain 'active' and a new start delay could start playback again at a later 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

Every channel and channel group has its own DSP Clock. A channel or channel group can be delayed relatively against its parent, with sample accurate positioning. To delay a sound, use the 'parent' channel group DSP clock to reference against when passing values into this function.

If a parent channel group changes its pitch, the start and stop times will still be correct as the parent clock is rate adjusted by that pitch.
See Also

- ChannelControl::getDelay
- ChannelControl::getDSPClock
- ChannelControl::isPlaying
Firelight Technologies FMOD Studio API
Channel::setFadePointRamp

Add a short 64 sample volume ramp to the specified time in the future using fade points.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setFadePointRamp(
    unsigned long long dspclock,
    float volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetFadePointRamp(
    FMOD_CHANNEL *channel,
    unsigned long long dspclock,
    float volume
);
```

**C# Syntax**

```csharp
RESULT Channel.setFadePointRamp(
    ulong dspclock,
    float volume
);
```

**JavaScript Syntax**

```javascript
Channel.setFadePointRamp(
    dspclock,
    volume
);
```
Parameters

dspclock
DSP clock of the parent channel group when the volume will be ramped to.

volume
Volume level where 0 is silent and 1.0 is normal volume. Amplification is supported.
Return Values

If the function succeeds then the return value 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 helper function that automatically ramps from the current fade volume to the newly provided volume at a specified time. It will clear any fade points set after this time. Use in conjunction with `ChannelControl::setDelay` stop delay, to ramp down volume before it stops. The user would specify the same clock value for the fade ramp and stop delay. This can also be used as a way to provide sample accurate delayed volume changes without clicks.
See Also

- `ChannelControl::setDelay`
- `ChannelControl::removeFadePoints`
- `ChannelControl::addFadePoint`
- `ChannelControl::getFadePoints`
Firelight Technologies FMOD Studio API
Channel::setFrequency

Sets the channel frequency or playback rate, in Hz.

C++ Syntax

```cpp
FMOD_RESULT Channel::setFrequency(
    float frequency
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetFrequency(
    FMOD_CHANNEL *channel,
    float frequency
);
```

C# Syntax

```csharp
RESULT Channel.setFrequency(
    float frequency
);
```

JavaScript Syntax

```javascript
Channel.setFrequency(
    frequency
);
```
Parameters

frequency

Frequency value in Hz. This value can also be negative to play the sound backwards (negative frequencies allowed with 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
- Sound::setDefaults
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT Channel.setLoopCount(
    int loopcount
);
```

JavaScript Syntax

```javascript
Channel.setLoopCount(
    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

*Issues with streamed audio:*

When changing the loop count, sounds created with `System::createStream` or `FMOD_CREATESTREAM` may have already 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 reflush the stream buffer by calling `Channel::setPosition`. Note this will usually only happen if you have sounds or loop points that are smaller than the stream decode buffer size.
See Also

- Channel::getLoopCount
- Channel::setPosition
- System::createStream
- System::setStreamBufferSize
- FMOD_CREATEBOUNDDEXINFO
- FMOD_MODE
Firelight Technologies FMOD Studio API
Channel::setLoopPoints

Sets the loop points within the channel.

C++ Syntax

FMOD_RESULT Channel::setLoopPoints(
    unsigned int loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int loopend,
    FMOD_TIMEUNIT loopendtype
);

C Syntax

FMOD_RESULT FMOD_Channel_SetLoopPoints(
    FMOD_CHANNEL *channel,
    unsigned int loopstart,
    FMOD_TIMEUNIT loopstarttype,
    unsigned int loopend,
    FMOD_TIMEUNIT loopendtype
);

C# Syntax

RESULT Channel.setLoopPoints(
    uint loopstart,
    TIMEUNIT loopstarttype,
    uint loopend,
    TIMEUNIT loopendtype
);

JavaScript Syntax

Channel.setLoopPoints(
    loopstart,
    loopstarttype,
    loopend,
    loopendtype
);
_parameters_

_loopstart_
Loop start point, this point in time is played so it is inclusive.

_loopstarttype_
Time format used for the loop start point (see `FMOD_TIMEUNIT`).

_loopend_
Loop end point, this point in time is played so it is inclusive.

_loopendtype_
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

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.

Issues with streamed audio:

When changing the loop count, sounds created with System::createStream or FMOD_CREATESTREAM may have already 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 by calling Channel::setPosition. Note this will usually only happen if you have sounds or loop points that are smaller than the stream decode buffer size.
See Also

- Channel::getLoopPoints
- Channel::setPosition
- System::createStream
- System::setStreamBufferSize
- FMOD_CREATEsoundEXINFO
- FMOD_MODE
- FMOD_TIMEUNIT
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Channel.setLowPassGain(
    float gain
);
```

**JavaScript Syntax**

```javascript
Channel.setLowPassGain(
    gain
);
```
Parameters

gain
Linear gain level, from 0 (silent, full filtering) to 1.0 (full volume, no filtering), 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

Requires the built in lowpass to be created with `FMOD_INIT_CHANNEL_LOWPASS` or `FMOD_INIT_CHANNEL_DISTANCEFILTER`.
See Also

- [ChannelControl::getLowPassGain](#)
Firelight Technologies FMOD Studio API
Channel::setMixLevelsInput

Sets the incoming volume level for each channel of a multi-channel sound. This is a helper to avoid calling ChannelControl::setMixMatrix.

A multi-channel sound is a single sound that contains from 1 to 32 channels of sound data, in an interleaved fashion. If in the extreme case, a 32ch wave file was used, an array of 32 floating point numbers denoting their volume levels would be passed in to the levels parameter, and 32 for the numlevels parameter.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setMixLevelsInput(
    float *levels,
    int numlevels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetMixLevelsInput(
    FMOD_CHANNEL *channel,
    float *levels,
    int numlevels
);
```

**C# Syntax**

```csharp
RESULT Channel.setMixLevelsInput(
    float[] levels,
    int numlevels
);
```

**JavaScript Syntax**

```javascript
Channel.setMixLevelsInput(
    levels,
    numlevels
);
```
Parameters

levels
   Array of volume levels for each incoming channel.
numlevels
   Number of levels in the array, from 0 to 32 inclusive.
**Return Values**

If the function succeeds then the return value 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 example use case for this function is if the sound file has multiple channels in it with different musical parts to it, but they are all in sync with each other. This function can be used to fade in and out different tracks of the sound or to solo/mute tracks within it.

NOTE: This function overwrites any pan/output mixlevel by overwriting the ChannelControl's matrix if it exists. It will create an NxN matrix where the output levels are the same as the input levels. If you wish to fold this down to a lower channel count mix rather than staying at the input channel count, either create a custom matrix instead and use ChannelControl::setMixMatrix, or add a new DSP after the fader, that has a different channel format (ie with ChannelControl::getDSP and DSP::setChannelFormat).

NOTE: Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.
See Also

- ChannelControl::setMixMatrix
- ChannelControl::getMixMatrix
- ChannelControl::getDSP
- DSP::setChannelFormat
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
Channel::setMixLevelsOutput

Sets the speaker volume levels for each speaker individually, this is a helper to avoid calling ChannelControl::setMixMatrix.

C++ Syntax

```cpp
FMOD_RESULT Channel::setMixLevelsOutput(
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float surroundleft,
    float surroundright,
    float backleft,
    float backright
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetMixLevelsOutput(
    FMOD_CHANNEL *channel,
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float surroundleft,
    float surroundright,
    float backleft,
    float backright
);
```

C# Syntax

```csharp
RESULT Channel.setMixLevelsOutput(
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float surroundleft,
    float surroundright,
    float backleft,
```
float backright
);

JavaScript Syntax

Channel.setMixLevelsOutput(
  frontleft,
  frontright,
  center,
  lfe,
  surroundleft,
  surroundright,
  backleft,
  backright
);

Parameters

frontleft
   Volume level for the front left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

frontright
   Volume level for the front right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

center
   Volume level for the center speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

lfe
   Volume level for the subwoofer speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

surroundleft
   Volume level for the surround left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

surroundright
   Volume level for the surround right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

backleft
   Volume level for the back left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).

backright
   Volume level for the back right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (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

NOTE: This function overwrites any pan/mixlevel by overwriting the ChannelControl's Matrix.

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion. 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 ChannelControl::setMixMatrix.
See Also

- `ChannelControl::setMixMatrix`
- `ChannelControl::getMixMatrix`
Firelight Technologies FMOD Studio API
**Channel::setMixMatrix**

Sets a 2D pan matrix that maps input channels (columns) to output speakers (rows).

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setMixMatrix(
    float *matrix,
    int outchannels,
    int inchannels,
    int matrixhop
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetMixMatrix(
    FMOD_CHANNEL *channel,
    float *matrix,
    int outchannels,
    int inchannels,
    int matrixhop
);
```

**C# Syntax**

```csharp
RESULT Channel.setMixMatrix(
    float[] matrix,
    int outchannels,
    int inchannels,
    int inchannel_hop
);
```

**JavaScript Syntax**

```javascript
Channel.setMixMatrix(
    matrix,
    outchannels,
    inchannels,
    inchannel_hop
);
```
Parameters

matrix
Address of a 2 dimensional array of volume levels in row-major order. Each row represents an output speaker, each column represents an input channel.

outchannels
Number of output channels (rows) in the matrix being passed in, from 0 to \text{FMOD\_MAX\_CHANNEL\_WIDTH} inclusive.

inchannels
Number of input channels (columns) in the matrix being passed in, from 0 to \text{FMOD\_MAX\_CHANNEL\_WIDTH} inclusive.

matrixhop
The width (total number of columns) of the matrix. Optional. If this is 0, inchannels will be taken as the width of the matrix. Maximum of \text{FMOD\_MAX\_CHANNEL\_WIDTH}.
Return Values

If the function succeeds then the return value 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 gain for input channel 's' to output channel 't' is matrix[t * matrixhop + s].

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

The matrix size will generally be the size of the number of channels in the current speaker mode. Use System::getSoftwareFormat to determine this.

If a matrix already exists then the matrix passed in will applied over the top of it. The input matrix can be smaller than the existing matrix.

A 'unit' matrix allows a signal to pass through unchanged. For example for a 5.1 matrix a unit matrix would look like this:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]
See Also

- ChannelControl::getMixMatrix
- ChannelControl::setPan
- ChannelControl::setMixLevelsOutput
- ChannelControl::setMixLevelsInput
- System::getSoftwareFormat
- FMOD_MAX_CHANNEL_WIDTH
Firelight Technologies FMOD Studio API
Channel::setMode

Changes some attributes for a channel or channelgroup based on the mode passed in.

C++ Syntax

FMOD_RESULT Channel::setMode(
    FMOD_MODE mode
);

C Syntax

FMOD_RESULT FMOD_Channel_SetMode(
    FMOD_CHANNEL *channel,
    FMOD_MODE mode
);

C# Syntax

RESULT Channel.setMode(
    MODE mode
);

JavaScript Syntax

Channel.setMode(
    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`
- `FMOD_2D`
- `FMOD_3D`
- `FMOD_3D_HEADRELATIVE`
- `FMOD_3D_WORLDRELATIVE`
- `FMOD_3D_INVERSEROLLOFF`
- `FMOD_3D_LINEARROLLOFF`
- `FMOD_3D_LINEARSQUAREROLLOFF`
- `FMOD_3D_CUSTOMROLLOFF`
- `FMOD_3D_IGNOREGEOMETRY`
- `FMOD_VIRTUAL_PLAYFROMSTART`

Issues with streamed audio:

When changing the loop mode, sounds created with `System::createStream` or `FMOD_CREATESTREAM` may have already 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 reflush the stream buffer by calling `Channel::setPosition`. Note this will usually only happen if you have sounds or loop points that are smaller than the stream decode buffer size.

Issues with PCM samples:

When changing the loop mode of sounds created with with `System::createSound` or `FMOD_CREATESAMPLE`, 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 (i.e. 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` or `FMOD_VIRTUAL_PLAYFROMSTART` is not specified, the flag will be cleared if it was specified previously.
See Also

- `Channel::getMode`
- `Channel::setPosition`
- `Sound::setMode`
- `System::createStream`
- `System::createSound`
- `System::setStreamBufferSize`
- `FMOD_CREATESOUNDEXINFO`
- `FMOD_MODE`
Channel::setMute

Sets the mute state 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  
);  
```

**C# Syntax**

```csharp
RESULT Channel.setMute(  
    bool mute  
);  
```

**JavaScript Syntax**

```javascript
Channel.setMute(  
    mute  
);  
```
Parameters

mute
Mute state, true = mute (silent), false = 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

Each channel and channel group has its own mute state, muting a channel group will mute all child channels but will not affect their individual setting. Calling `ChannelControl::getMute` will always return the value you set.
See Also

- ChannelControl::getMute
Firelight Technologies FMOD Studio API
Channel::setPan

Sets the pan level, this is a helper to avoid calling ChannelControl::setMixMatrix.

C++ Syntax

```cpp
FMOD_RESULT Channel::setPan(
    float pan
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetPan(
    FMOD_CHANNEL *channel,
    float  pan
);
```

C# Syntax

```csharp
RESULT Channel.setPan(
    float pan
);
```

JavaScript Syntax

```javascript
Channel.setPan(
    pan
);
```
Parameters

pan

Pan level, from -1.0 (left) to 1.0 (right), default = 0 (center).
Return Values

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

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

- ChannelControl::setMixMatrix
- ChannelControl::getMixMatrix
Firelight Technologies FMOD Studio API
Channel::setPaused

Sets the paused state.

C++ Syntax

```cpp
FMOD_RESULT Channel::setPaused(
    bool paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_Channel_SetPaused(
    FMOD_CHANNEL *channel,
    FMOD_BOOL paused
);
```

C# Syntax

```csharp
RESULT Channel.setPaused(
    bool paused
);
```

JavaScript Syntax

```javascript
Channel.setPaused(
    paused
);
```
Parameters

paused
  Paused state, 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

Each channel and channel group has its own paused state, pausing a channel group will pause all contained channels but will not affect their individual setting. Calling `ChannelControl::getPaused` will always return the value you set.
See Also

- ChannelControl::getPaused
Firelight Technologies FMOD Studio API
Channel::setPitch

Sets the pitch value.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setPitch(
    float pitch
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetPitch(
    FMOD_CHANNEL *channel,
    float pitch
);
```

**C# Syntax**

```csharp
RESULT Channel.setPitch(
    float pitch
);
```

**JavaScript Syntax**

```javascript
Channel.setPitch(
    pitch
);
```
Parameters

pitch
Pitch value, 0.5 = half pitch, 2.0 = double pitch, etc 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 scales existing frequency values by the pitch.
See Also

- ChannelControl::getPitch
- Channel::setFrequency
- Channel::getFrequency
Firelight Technologies FMOD Studio API
Channel::setPosition

Sets the playback position for the currently playing sound to the specified 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
);
```

C# Syntax

```csharp
RESULT Channel.setPosition(
    uint position,
    TIMEUNIT postype
);
```

JavaScript Syntax

```javascript
Channel.setPosition(
    position,
    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.

If you are calling this function on a stream, it has to possibly reflush 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::getOpenState`
Firelight Technologies FMOD Studio API
Channel::setPriority

Sets the priority for the channel after it has been played.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setPriority(
    int priority
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetPriority(
    FMOD_CHANNEL *channel,
    int priority
);
```

**C# Syntax**

```csharp
RESULT Channel.setPriority(
    int priority
);
```

**JavaScript Syntax**

```javascript
Channel.setPriority(
    priority
);
```
Parameters

priority
Priority for the channel, from 0 (most important) to 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

When more channels than available are played the virtual channel system will choose existing channels to steal. Lower priority sounds will always be stolen before higher priority sounds. For channels of equal priority, that with the quietest `ChannelControl::getAudibility` value will be stolen.
See Also

- Channel::getPriority
- ChannelControl::getAudibility
Firelight Technologies FMOD Studio API
Channel::setReverbProperties

Sets the wet level (or send level) of a particular reverb instance.

C++ Syntax

FMOD_RESULT Channel::setReverbProperties(
    int instance,
    float wet
);

C Syntax

FMOD_RESULT FMOD_Channel_SetReverbProperties(
    FMOD_CHANNEL *channel,
    int instance,
    float wet
);

C# Syntax

RESULT Channel.setReverbProperties(
    int instance,
    float wet
);

JavaScript Syntax

Channel.setReverbProperties(
    instance,
    wet
);
Parameters

instance
   Index of the particular reverb instance to target, from 0 to
   FMOD_REVERB_MAXINSTANCES inclusive.

wet
   Send level for the signal to the reverb, from 0 (none) to 1.0 (full), default =
   1.0 for Channels, 0.0 for ChannelGroups. 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

A Channel is automatically connected to all existing reverb instances due to the default wet level of 1.0. A ChannelGroup however will not send to any reverb by default requiring an explicit call to this function.

A ChannelGroup reverb is optimal for the case where you want to send 1 mixed signal to the reverb, rather than a lot of individual channel reverb sends. It is advisable to do this to reduce CPU if you have many Channels inside a ChannelGroup.

Keep in mind when setting a wet level for a ChannelGroup, any Channels under that ChannelGroup will still have their existing sends to the reverb. To avoid this doubling up you should explicitly set the Channel wet levels to 0.0.
See Also

- ChannelControl::getReverbProperties
Firelight Technologies FMOD Studio API
Channel::<setUserData

Sets a user value that can be retrieved with ChannelControl::<getUserData.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::<setUserData(
    void *userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Channel_SetUserData(
    FMOD_CHANNEL  *channel,
    void *userdata
);
```

**C# Syntax**

```csharp
RESULT Channel.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
Channel.setUserData(
    userdata
);
```

Parameters

userdata
  Data that the user wishes stored 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

You can use this to store a pointer to any wrapper or internal class that is associated with this object. This is especially useful for callbacks where you need to get back access to your own objects.
See Also

- ChannelControl::getUserData
Firelight Technologies FMOD Studio API
Channel::setVolume

Sets the volume level linearly.

**C++ Syntax**

```cpp
FMOD_RESULT Channel::setVolume(
    float volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD.Channel_SetVolume(
    FMOD_CHANNEL *channel,
    float volume
);
```

**C# Syntax**

```csharp
RESULT Channel.setVolume(
    float volume
);
```

**JavaScript Syntax**

```javascript
Channel.setVolume(
    volume
);
```
Parameters

volume
Linear volume level, 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

Volume level can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

Sound::setDefaults can be used to change the default volume for any channels played using that sound.
See Also

- `ChannelControl::getVolume`
- `Sound::setDefaults`
Firelight Technologies FMOD Studio API
Channel::setVolumeRamp

Sets whether the channel automatically ramps when setting volumes.

C++ Syntax

FMODRESULT Channel::setVolumeRamp(
    bool ramp
);

C Syntax

FMOD_RESULT FMOD_Channel_SetVolumeRamp(
    FMOD_CHANNEL *channel,  
    FMOD_BOOL ramp
);

C# Syntax

RESULT Channel.setVolumeRamp(
    bool ramp
);

JavaScript Syntax

Channel.setVolumeRamp(
    ramp
);
Parameters

ramp
   Whether to enable volume ramping.
Return Values

If the function succeeds then the return value 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 changing volumes on a non-paused channel, FMOD normally adds a small ramp to avoid a pop sound. This function allows that setting to be overridden and volume changes to be applied immediately.
See Also

- [ChannelControl::getVolumeRamp](#)
Firelight Technologies FMOD Studio API
Channel::stop

Stops the channel (or all channels in the channel group) 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);
```

**C# Syntax**

```csharp
RESULT Channel.stop();
```

**JavaScript Syntax**

```javascript
Channel.stop();
```
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

- System::playSound
Firelight Technologies FMOD Studio API
ChannelGroup
Functions

ChannelGroup::addDSP  ChannelGroup::addFadePoint
ChannelGroup::addGroup
ChannelGroup::get3DAttributes
ChannelGroup::get3DConeOrientation
ChannelGroup::get3DConeSettings
ChannelGroup::get3DCustomRolloff
ChannelGroup::get3DDistanceFilter
ChannelGroup::get3DDopplerLevel
ChannelGroup::get3DLevel
ChannelGroup::get3DMinMaxDistance
ChannelGroup::get3DOcclusion
ChannelGroup::get3DSpread
ChannelGroup::getAudibility
ChannelGroup::getChannel
ChannelGroup::getDSP
ChannelGroup::getDSPClock
ChannelGroup::getDSPIndex
ChannelGroup::getDelay
ChannelGroup::getFadePoints
ChannelGroup::getGroup
ChannelGroup::getLowPassGain
ChannelGroup::getMixMatrix
ChannelGroup::getMode
ChannelGroup::getMute
ChannelGroup::getName
ChannelGroup::getNumChannels
ChannelGroup::getNumDSPs
ChannelGroup::getNumGroups
ChannelGroup::getParentGroup
ChannelGroup::getPaused
ChannelGroup::getPitch
ChannelGroup::getReverbProperties
ChannelGroup::getSystemObject
ChannelGroup::getUserData
ChannelGroup::getVolume
ChannelGroup::getVolumeRamp
ChannelGroup::isPlaying
ChannelGroup::release
ChannelGroup::removeDSP
ChannelGroup::removeFadePoints
ChannelGroup::set3DAttributes
ChannelGroup::set3DConeOrientation
ChannelGroup::set3DConeSettings
ChannelGroup::set3DCustomRolloff
ChannelGroup::set3DDistanceFilter
ChannelGroup::set3DDopplerLevel
ChannelGroup::set3DLevel
ChannelGroup::set3DMinMaxDistance
ChannelGroup::set3DOcclusion
ChannelGroup::set3DSpread
ChannelGroup::setCallback
ChannelGroup::setDSPIndex
ChannelGroup::setDelay
ChannelGroup::setFadePointRamp
ChannelGroup::setLowPassGain
ChannelGroup::setMixLevelsInput
ChannelGroup::setMixLevelsOutput
ChannelGroup::setMixMatrix
ChannelGroup::setMode
ChannelGroup::setMute
ChannelGroup::setPan
ChannelGroup::setPaused
ChannelGroup::setPitch
ChannelGroup::setReverbProperties
ChannelGroup::setUserData
ChannelGroup::setVolume
ChannelGroup::setVolumeRamp
ChannelGroup::stop
Firelight Technologies FMOD Studio API
**ChannelGroup::addDSP**

Add a pre-created DSP unit to the specified index in the DSP chain.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::addDSP(
    int index,
    DSP *dsp
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_AddDSP(
    FMOD_CHANNELGROUP *channelgroup,
    int index,
    FMOD_DSP *dsp
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.addDSP(
    int index,
    DSP dsp
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.addDSP(
    index,
    dsp
);
```
Parameters

index
   Offset to add this DSP unit at in the DSP chain, see
   `FMOD_CHANNELCONTROL_DSP_INDEX` for special named offsets.

dsp
   Pointer to a pre-created DSP unit to be inserted at the specified offset.
Return Values

If the function succeeds then the return 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::createDSPByPlugin
- ChannelControl::removeDSP
- ChannelControl::getDSP
- ChannelControl::getNumDSPs
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
ChannelGroup::addFadePoint

Add a volume point to fade from or towards, using a clock offset and 0 to 1 volume level.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::addFadePoint(
    unsigned long long dspclock,
    float volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_AddFadePoint(
    FMOD_CHANNELGROUP *channelgroup,
    unsigned long long dspclock,
    float volume
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.addFadePoint(
    ulong dspclock,
    float volume
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.addFadePoint(
    dspclock,
    volume
);
```

Parameters

dspclock
   DSP clock of the parent channel group to set the fade point volume.
volume
   Volume level where 0 is silent and 1.0 is normal volume. Amplification is supported.
Return Values

If the function succeeds then the return value 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 every fade point, FMOD will do a per sample volume ramp between them. It will scale with the current Channel or ChannelGroup's volume.

// Ramp from full volume to half volume over the next 4096 samples
FMOD_RESULT result;
unsigned long long parentclock;
result = target->getDSPClock(NULL, &parentclock);
result = target->addFadePoint(parentclock, 1.0f);
result = target->addFadePoint(parentclock + 4096, 0.5f);
See Also

- ChannelControl::removeFadePoints
- ChannelControl::setFadePointRamp
- ChannelControl::getFadePoints
Firelight Technologies FMOD Studio API
ChannelGroup::addGroup

Adds a channel group as a child of the current channel group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::addGroup(
    FMOD::ChannelGroup *group,
    bool propagatedspclock,
    FMOD::DSPConnection **connection
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_AddGroup(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_CHANNELGROUP *group,
    FMOD_BOOL propagatedspclock,
    FMOD_DSPCONNECTION **connection
);
```

C# Syntax

```csharp
RESULT ChannelGroup.addGroup(
    ChannelGroup group,
    bool propagatedspclock,
    out DSPConnection connection
);
```

JavaScript Syntax

```javascript
ChannelGroup.addGroup(
    group,
    propagatedspclock,
    connection // writes value to connection.val
);
```
Parameters

**group**
Channel group to add as a child.

**propagatedspclock**
When a child group is added to a parent group, the clock values from the parent will be propagated down into the child.

**connection**
Address of a variable to receive a pointer to a DSP connection, which is the connection between the parent and the child group's DSP 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.
See Also

- ChannelGroup::getNumGroups
- ChannelGroup::getGroup
- ChannelGroup::getParentGroup
Firelight Technologies FMOD Studio API
ChannelGroup::get3DAttributes

Retrieves the position and velocity used to apply panning, attenuation and doppler.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::get3DAttributes(
    FMOD_VECTOR *pos,
    FMOD_VECTOR *vel,
    FMOD_VECTOR *alt_pan_pos
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_Get3DAttributes(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_VECTOR *pos,
    FMOD_VECTOR *vel,
    FMOD_VECTOR *alt_pan_pos
);
```

C# Syntax

```csharp
RESULT ChannelGroup.get3DAttributes(
    out VECTOR pos,
    out VECTOR vel,
    out VECTOR alt_pan_pos
);
```

JavaScript Syntax

```javascript
ChannelGroup.get3DAttributes(
    pos, // writes value to pos.val
    vel, // writes value to vel.val
    alt_pan_pos // writes value to alt_pan_pos.va
);
```
Parameters

pos
Address of a variable that receives the position in 3D space used for panning and attenuation. Optional, specify 0 or NULL to ignore.

vel
Address of a variable that receives the velocity in 'distance units per second' (see remarks) in 3D space. Optional, specify 0 or NULL to ignore.

alt_pan_pos
(Unimplemented).
Return Values

If the function succeeds then the return value 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

- ChannelControl::set3DAttributes
- System::set3DSettings
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
ChannelGroup::get3DConeOrientation

Retrieves the orientation of the sound projection cone.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::get3DConeOrientation(
    FMOD_VECTOR *orientation
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Get3DConeOrientation(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_VECTOR *orientation
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.get3DConeOrientation(
    out VECTOR orientation
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.get3DConeOrientation(
    orientation // writes value to orientation.val
);
```
**Parameters**

orientation  
Address of a variable that receives the coordinates of the sound cone orientation vector, 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

- ChannelControl::set3DConeOrientation
Firelight Technologies FMOD Studio API
**ChannelGroup::get3DConeSettings**

Retrieves the angles that define the sound projection cone including the volume when outside the cone.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::get3DConeSettings(
    float *insideconeangle,
    float *outsideconeangle,
    float *outsidevolume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Get3DConeSettings(
    FMOD_CHANNELGROUP *channelgroup,
    float *insideconeangle,
    float *outsideconeangle,
    float *outsidevolume
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.get3DConeSettings(
    out float insideconeangle,
    out float outsideconeangle,
    out float outsidevolume
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.get3DConeSettings(
    insideconeangle, // writes value to insideconeangle
    outsideconeangle, // writes value to outsideconeangle
    outsidevolume // writes value to outsidevolume.
);
```
Parameters

insideconeangle
Address of a variable that receives the inside cone angle, in degrees. This is the angle within which the sound is at its normal volume.

outsideconeangle
Address of a variable that receives the outside cone angle, in degrees. This is the angle outside of which the sound is at its outside volume.

outsidevolume
Address of a variable that receives the cone outside 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

- [ChannelControl::set3DConesSettings](#)
Firelight Technologies FMOD Studio API
ChannelGroup::get3DCustomRolloff

Retrieves a pointer to the current custom rolloff curve.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::get3DCustomRolloff(
    FMOD_VECTOR **points,
    int *numpoints
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Get3DCustomRolloff(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_VECTOR **points,
    int *numpoints
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.get3DCustomRolloff(
    out IntPtr points,
    out int numpoints
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.get3DCustomRolloff(
    points, // writes value to points.val
    numpoints // writes value to numpoints.val
);
```
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

- ChannelControl::set3DCustomRolloff
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
ChannelGroup::get3DDistanceFilter

Retrieve the settings for the 3D distance filter properties for a Channel or
Channel Group.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::get3DDistanceFilter(
    bool *custom,
    float *customLevel,
    float *centerFreq
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_Get3DDistanceFilter(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL *custom,
    float *customLevel,
    float *centerFreq
);
```

C# Syntax

```csharp
RESULT ChannelGroup.get3DDistanceFilter(
    out bool custom,
    out float customLevel,
    out float centerFreq
);
```

JavaScript Syntax

```javascript
ChannelGroup.get3DDistanceFilter(
    custom, // writes value to custom.val
    customLevel, // writes value to customLevel.val
    centerFreq // writes value to centerFreq.val
);
```
Parameters

custom
   Address of a variable to receive the enabled/disabled state of the FMOD
distance rolloff calculation. Default = false.

customLevel
   Address of a variable to receive the manual user attenuation, where 1.0 = no
attenuation and 0 = complete attenuation. Default = 1.0.

centerFreq
   Address of a variable to receive center frequency in hz for the high-pass
filter used to simulate distance attenuation, from 10.0 to 22050.0. Default =
1500.0.
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

- ChannelControl::set3DDistanceFilter
Firelight Technologies FMOD Studio API
ChannelGroup::get3DDopplerLevel

Retrieves the amount by which doppler is scaled.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::get3DDopplerLevel(
    float *level
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_Get3DDopplerLevel(
    FMOD_CHANNELGROUP *channelgroup,
    float *level
);
```

C# Syntax

```csharp
RESULT ChannelGroup.get3DDopplerLevel(
    out float level
);
```

JavaScript Syntax

```javascript
ChannelGroup.get3DDopplerLevel(
    level // writes value to level.val
);
```
Parameters

level
   Address of a variable to receive the doppler scale from 0.0 (none), to 1.0 (normal) to 5.0 (exaggerated).
Return Values

If the function succeeds then the return 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

- ChannelControl::set3DDopplerLevel
Firelight Technologies FMOD Studio API
ChannelGroup::get3DLevel

Retrieves the current 3D mix level set by ChannelControl::set3DPanLevel.

C++ Syntax

FMOD_RESULT ChannelGroup::get3DLevel(
    float *level
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_Get3DLevel(
    FMOD_CHANNELGROUP *channelgroup,
    float *level
);

C# Syntax

RESULT ChannelGroup.get3DLevel(
    out float level
);

JavaScript Syntax

ChannelGroup.get3DLevel(
    level // writes value to level.val
);
Parameters

level

3D pan level from 0.0 (attenuation is ignored and panning as set by 2D panning functions) to 1.0 (pan and attenuate according to 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

- ChannelControl::set3DPanLevel
Firelight Technologies FMOD Studio API
ChannelGroup::get3DMinMaxDistance

Retrieves the minimum and maximum audible distance.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::get3DMinMaxDistance(
    float *mindistance,
    float *maxdistance
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Get3DMinMaxDistance(
    FMOD_CHANNELGROUP *channelgroup,
    float *mindistance,
    float *maxdistance
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.get3DMinMaxDistance(
    out float mindistance,
    out float maxdistance
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.get3DMinMaxDistance(
    mindistance, // writes value to mindistance.val
    maxdistance // writes value to maxdistance.val
);
```
Parameters

mindistance
Address of a variable that receives the minimum volume distance in 'units' (see remarks). Optional, specify 0 or NULL to ignore.

maxdistance
Address of a variable that receives the maximum volume distance in 'units' (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

- ChannelControl::set3DMinMaxDistance
- System::set3DSettings
Firelight Technologies FMOD Studio API
ChannelGroup::get3DOcclusion

Retrieves the occlusion factors.

**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
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.get3DOcclusion(
    out float directocclusion,
    out float reverbocclusion
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.get3DOcclusion(
    directocclusion, // writes value to directocclusion
    reverbocclusion  // writes value to reverbocclusion
);
```
**Parameters**

directocclusion
   Address of a variable that receives the occlusion factor for the direct path.
reverbocclusion
   Address of a variable that receives the occlusion factor for the reverb mix.
Return Values

If the function succeeds then the return 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

- ChannelControl::set3DOcclusion
Firelight Technologies FMOD Studio API
ChannelGroup::get3DSpread

Retrieves the spread of a 3D sound in speaker space.

C++ Syntax

FMOD_RESULT ChannelGroup::get3DSpread(
    float *angle
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_Get3DSpread(
    FMOD_CHANNELGROUP *channelgroup,
    float *angle
);

C# Syntax

RESULT ChannelGroup.get3DSpread(
    out float angle
);

JavaScript Syntax

ChannelGroup.get3DSpread(
    angle
) // writes value to angle.val
Parameters

angle
Address of a variable that receives the speaker spread angle.
Return Values

If the function succeeds then the return 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

- ChannelControl::set3DSpread
Firelight Technologies FMOD Studio API
ChannelGroup::getAudibility

Retrieves the combined volume after 3D spatialization and geometry occlusion calculations including any volumes set via the API.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getAudibility(
    float *audibility
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetAudibility(
    FMOD_CHANNELGROUP *channelgroup,
    float *audibility
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getAudibility(
    out float audibility
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getAudibility(
    audibility // writes value to audibility.val
);
```
Parameters

audibility
   Address of a variable that receives the 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 result of all volume modifiers. This value is used by the virtual channel system to order its channels between real and virtual.

See the Virtual Voice System page for more details about how the audibility is calculated.
See Also

- Channel::isVirtual
- ChannelControl::getVolume
- ChannelControl::get3DOcclusion
- ChannelControl::get3DAttributes
- FMOD_DSP_PARAMETER_OVERALLGAIN
- FMOD_DSP_PARAMETER_DATA_TYPE_OVERALLGAIN
Firelight Technologies FMOD Studio API
ChannelGroup::getChannel

Retrieves the specified channel from the channel group.

C++ Syntax

FMOD_RESULT ChannelGroup::getChannel(
    int index,
    FMOD::Channel **channel
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_GetChannel(
    FMOD_CHANNELGROUP *channelgroup,
    int index,
    FMOD_CHANNEL **channel
);

C# Syntax

RESULT ChannelGroup.getChannel(
    int index,
    out Channel channel
);

JavaScript Syntax

ChannelGroup.getChannel(
    index,
    channel // writes value to channel.val
);
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.
Return Values

If the function succeeds then the return 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 Studio API
ChannelGroup::getDSP

Retrieve the DSP unit at the specified index.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getDSP(
    int index,
    DSP **dsp
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetDSP(
    FMOD_CHANNELGROUP *channelgroup,
    int index,
    FMOD_DSP **dsp
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getDSP(
    int index,
    out DSP dsp
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getDSP(
    index,
    dsp // writes value to dsp.val
);
```
Parameters

index
  Offset into the DSP chain, see
  FMOD_CHANNELCONTROL_DSP_INDEX for special named offsets.

dsp
  Address of a variable to receive a pointer to the requested DSP unit.
Return Values

If the function succeeds then the return value is $\text{FMOD\_OK}$.
If the function fails then the return value will be one of the values defined in the $\text{FMOD\_RESULT}$ enumeration.
See Also

- `ChannelControl::addDSP`
- `ChannelControl::removeDSP`
- `ChannelControl::getNumDSPs`
- `FMOD_CHANNELCONTROL_DSP_INDEX`
Firelight Technologies FMOD Studio API
ChannelGroup::getDSPClock

Retrieves the DSP clock values which count up by the number of samples per second in the software mixer, i.e. if the default sample rate is 48KHz, the DSP clock increments by 48000 per second.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getDSPClock(  
    unsigned long long *dspclock,  
    unsigned long long *parentclock
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetDSPClock(  
    FMOD_CHANNELGROUP *channelgroup,  
    unsigned long long *dspclock,  
    unsigned long long *parentclock
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getDSPClock(  
    out ulong dspclock,  
    out ulong parentclock
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getDSPClock(  
    dspclock, // writes value to dspclock.val  
    parentclock // writes value to parentclock.val
);
```
Parameters

dspclock
Address of a variable to receive the DSP clock value for the head DSP node.

parentclock
Address of a variable to receive the DSP clock value for the tail DSP node.
Return Values

If the function succeeds then the return value 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 ChannelControl::setDelay to play a sound on an exact tick in the future, or stop it in the future.

Note that when delaying a channel or channel group you want to sync it to the parent channel group DSP clock value, not its own DSP clock value.
See Also

- `ChannelControl::setDelay`
- `ChannelControl::getDelay`
Firelight Technologies FMOD Studio API
ChannelGroup::getDSPIndex

Retrieves the index in the DSP chain of the provided DSP.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getDSPIndex(
    DSP *dsp,
    int *index
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetDSPIndex(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_DSP *dsp,
    int *index
);
```

C# Syntax

```csharp
RESULT ChannelGroup.getDSPIndex(
    DSP dsp,
    out int index
);
```

JavaScript Syntax

```javascript
ChannelGroup.getDSPIndex(
    dsp,
    index // writes value to index.val
);```
Parameters

dsp
   Pointer to a DSP unit that exists in the DSP chain.

index
   Address of a variable to receive the offset in the DSP chain of the specified DSP.
Return Values

If the function succeeds then the return 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

- ChannelControl::setDSPIndex
Firelight Technologies FMOD Studio API
ChannelGroup::getDelay

Retrieves a start (and/or stop) time relative to the parent channel group DSP clock, with sample accuracy.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getDelay(
    unsigned long long *dspclock_start,
    unsigned long long *dspclock_end,
    bool *stopchannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetDelay(
    FMOD_CHANNELGROUP *channelgroup,
    unsigned long long *dspclock_start,
    unsigned long long *dspclock_end,
    FMOD_BOOL *stopchannels
);
```

C# Syntax

```csharp
RESULT ChannelGroup.getDelay(
    out ulong dspclock_start,
    out ulong dspclock_end,
    out bool stopchannels
);
```

JavaScript Syntax

```javascript
ChannelGroup.getDelay(
    dspclock_start, // writes value to dspclock_start
    dspclock_end,   // writes value to dspclock_end.v
    stopchannels    // writes value to stopchannels.v
);
```
Parameters

dspclock_start
Address of a variable that receives the DSP clock of the parent channel group to audibly start playing sound at. Optional, specify 0 or NULL to ignore.

dspclock_end
Address of a variable that receives the DSP clock of the parent channel group to audibly stop playing sound at. Optional, specify 0 or NULL to ignore.

stopchannels
Address of a variable that receives TRUE = stop according to ChannelControl::isPlaying. FALSE = remain 'active' and a new start delay could start playback again at a later time. Optional, specify 0 or NULL to ignore.
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

- `ChannelControl::setDelay`
- `ChannelControl::getDSPClock`
- `ChannelControl::isPlaying`
Firelight Technologies FMOD Studio API
ChannelGroup::getFadePoints

Retrieve information about fade points stored within a Channel or ChannelGroup.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getFadePoints(
    unsigned int *numpoints,
    unsigned long long *point_dspclock,
    float *point_volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetFadePoints(
    FMOD_CHANNELGROUP *channelgroup,
    unsigned int *numpoints,
    unsigned long long *point_dspclock,
    float *point_volume
);
```

C# Syntax

```csharp
RESULT ChannelGroup.getFadePoints(
    ref uint numpoints,
    ulong[] point_dspclock,
    float[] point_volume
);
```

JavaScript Syntax

```javascript
ChannelGroup.getFadePoints(
    numpoints, // writes value to numpoints.val
    point_dspclock, // writes value to point_dspclock
    point_volume // writes value to point_volume.v
);
```
Parameters

numpoints
   Address of a variable to receive the number of fade points stored within the Channel or ChannelGroup.
point_dspclock
   Address of a variable to receive an array of 64bit clock values. Can be 0 or NULL.
point_volume
   Address of a variable to receive an array of floating point volume values. Can be 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 `FMOD_RESULT` enumeration.
Remarks

To first get the number of points for memory purposes, and not store any data, call this function with point_dpsclock and point_volume parameters being 0 or NULL.
See Also

- ChannelControl::addFadePoint
- ChannelControl::removeFadePoints
Firelight Technologies FMOD Studio API
ChannelGroup::getGroup

Retrieves a handle to a specified sub channel group.

**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
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getGroup(
    int index,
    out ChannelGroup group
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getGroup(
    index,
    group // writes value to group.val
);
```
Parameters

index
   Index to specify which sub channel group to receive.

   Address of a variable to receive a pointer to a 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::addGroup
- ChannelGroup::getNumGroups
- ChannelGroup::getParentGroup
Firelight Technologies FMOD Studio API
ChannelGroup::getLowPassGain

Retrieves the gain of the dry signal when lowpass filtering is applied.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getLowPassGain(
    float *gain
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetLowPassGain(
    FMOD_CHANNELGROUP *channelgroup,
    float *gain
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getLowPassGain(
    out float gain
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getLowPassGain(
    gain // writes value to gain.val
);
```
Parameters

gain
Address of a variable that receives the linear gain level, from 0 (silent) to 1.0 (full volume).
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

- ChannelControl::setLowPassGain
Firelight Technologies FMOD Studio API
ChannelGroup::getMixMatrix

Retrieves a 2D pan matrix that maps input channels (columns) to output speakers (rows).

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::getMixMatrix(
    float *matrix,
    int *outchannels,
    int *inchannels,
    int matrixhop
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_GetMixMatrix(
    FMOD_CHANNELGROUP *channelgroup,
    float *matrix,
    int *outchannels,
    int *inchannels,
    int matrixhop
);
```

C# Syntax

```csharp
RESULT ChannelGroup.getMixMatrix(
    float[] matrix,              // writes value to matrix.val
    out int outchannels,        // writes value to outchannels.val
    out int inchannels,         // writes value to inchannels.val
    int inchannel_hop
);
```

JavaScript Syntax

```javascript
ChannelGroup.getMixMatrix(
    matrix,                      // writes value to matrix.val
    outchannels,                 // writes value to outchannels.val
    inchannels,                  // writes value to inchannels.val
    inchannel_hop
);
```
**Parameters**

**matrix**
Address of a 2 dimensional array of volume levels in row-major order. Each row represents an output speaker, each column represents an input channel.

**outchannels**
Address of a variable to receive the number of output channels (rows) in the matrix being passed in.

**inchannels**
Address of a variable to receive the number of input channels (columns) in the matrix being passed in.

**matrixhop**
The width (total number of columns) of the matrix. Optional. If this is 0, inchannels will be taken as the width of the matrix. Maximum of **FMOD_MAX_CHANNEL_WIDTH**.
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

The gain for input channel 's' to output channel 't' is matrix[t * matrixhop + s].

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

The matrix size will generally be the size of the number of channels in the current speaker mode. Use System::getSoftwareFormat to determine this.

Passing NULL for 'matrix' will allow you to query 'outchannels' and 'inchannels' without copying any data.
See Also

- ChannelControl::setMixMatrix
- System::getSoftwareFormat
- FMOD_MAX_CHANNEL_WIDTH
Firelight Technologies FMOD Studio API
ChannelGroup::getMode

Retrieves the mode bit flags for the channel.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getMode(
    FMOD_MODE *mode
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetMode(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_MODE *mode
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getMode(
    out MODE mode
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getMode(
    mode
                        // writes value to mode.val
);
```
**Parameters**

*mode*

Address of a variable to receive the mode bits.
Return Values

If the function succeeds then the return 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
- FMOD_MODE
Firelight Technologies FMOD Studio API
ChannelGroup::getMute

Retrieves the mute state.

C++ Syntax

FMOD_RESULT ChannelGroup::getMute(
    bool *mute
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_GetMute(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL *mute
);

C# Syntax

RESULT ChannelGroup.getMute(
    out bool mute
);

JavaScript Syntax

ChannelGroup.getMute(
    mute // writes value to mute.val
);
Parameters

mute
   Address of a variable that receives the current mute state, true = mute (silent), false = 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

- ChannelControl::setMute
Firelight Technologies FMOD Studio API
ChannelGroup::getName

Retrieves the name of the channel group set when the group was 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
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getName(
    StringBuilder name,
    int namelen
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getName(
    name                                              // writes value to name.val
);
```
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.
Remarks

JavaScript only:

Note: For the "name" parameter, the maximum string length is 512.
See Also

- System::getMasterChannelGroup
- System::createChannelGroup

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
ChannelGroup::getNumChannels

Retrieves the 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
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getNumChannels(
    out int numchannels
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getNumChannels(
    numchannels // writes value to numchannels.var
);
```
Parameters

numchannels
    Address of a variable to receive the current number of 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 Studio API
ChannelGroup::getNumDSPs

Retrieves the number of DSP units in the DSP chain.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getNumDSPs(
    int *numdsps
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetNumDSPs(
    FMOD_CHANNELGROUP *channelgroup,
    int *numdsps
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getNumDSPs(
    out int numdsps
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getNumDSPs(
    numdsps // writes value to numdsps.val
);
```
Parameters

numdsp
Address of a variable that receives the number of DSP units in the DSP chain.
Return Values

If the function succeeds then the return 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

- `ChannelControl::addDSP`
- `ChannelControl::removeDSP`
- `ChannelControl::getDSP`
Firelight Technologies FMOD Studio API
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  
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getNumGroups(  
    out int numgroups  
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getNumGroups(  
    numgroups // writes value to numgroups.val  
);
```
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::addGroup`
- `ChannelGroup::getGroup`
- `ChannelGroup::getParentGroup`
Firelight Technologies FMOD Studio API
ChannelGroup::getParentGroup

Retrieves a handle to the channel group parent.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getParentGroup(
    FMOD::ChannelGroup **group
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetParentGroup(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_CHANNELGROUP **group
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getParentGroup(
    out ChannelGroup group
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getParentGroup(
    group
    // writes value to group.val
);
```
Parameters

group
   Address of a variable to receive a pointer to a 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::addGroup
- ChannelGroup::getNumGroups
- ChannelGroup::getGroup
Firelight Technologies FMOD Studio API
ChannelGroup::getPaused

Retrieves the paused state.

C++ Syntax

FMOD_RESULT ChannelGroup::getPaused(
    bool *paused
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_GetPaused(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL *paused
);

C# Syntax

RESULT ChannelGroup.getPaused(
    out bool paused
);

JavaScript Syntax

ChannelGroup.getPaused(
    paused
) // writes value to paused.val

);
Parameters

paused
   Address of a variable that receives the current paused state, true = paused, false = 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

- ChannelControl::setPaused
Firelight Technologies FMOD Studio API
ChannelGroup::getPitch

Retrieves the pitch value.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getPitch(
    float *pitch
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetPitch(
    FMOD_CHANNELGROUP *channelgroup,
    float *pitch
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getPitch(
    out float pitch
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getPitch(
    pitch // writes value to pitch.val
);
```
Parameters

pitch
   Address of a variable to receive the pitch value, 0.5 = half pitch, 2.0 = double pitch, etc.
Return Values

If the function succeeds then the return 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

- `ChannelControl::setPitch`
- `Channel::setFrequency`
- `Channel::getFrequency`
Firelight Technologies FMOD Studio API
ChannelGroup::getReverbProperties

Retrieves the wet level (or send level) for a particular reverb instance.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getReverbProperties(
    int instance,
    float *wet
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetReverbProperties(
    FMOD_CHANNELGROUP *channelgroup,
    int instance,
    float *wet
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getReverbProperties(
    int instance,
    out float wet
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getReverbProperties(
    instance,
    wet // writes value to wet.val
);
```
Parameters

instance
   Index of the particular reverb instance to target, from 0 to
   \texttt{FMOD\_REVERB\_MAXINSTANCES} inclusive.

wet
   Address of a variable that receives the send level for the signal to the
   reverb, from 0 (none) to 1.0 (full).
Return Values

If the function succeeds then the return 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

- ChannelControl::setReverbProperties
Firelight Technologies FMOD Studio API
ChannelGroup::getSystemObject

Retrieves the parent System object that created the channel or channel group.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getSystemObject(
    System **system
);
```

**C Syntax**

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

**C# Syntax**

```csharp
RESULT ChannelGroup.getSystemObject(
    out System system
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getSystemObject(
    system // writes value to system.val
);
```
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`
- `System::playSound`
Firelight Technologies FMOD Studio API
ChannelGroup::getUserData

Retrieves a user value that can be set with ChannelControl::setUserData.

C++ Syntax

FMOD_RESULT ChannelGroup::getUserData(
    void **userdata
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_GetUserData(
    FMOD_CHANNELGROUP *channelgroup,
    void **userdata
);

C# Syntax

RESULT ChannelGroup.getUserData(
    out IntPtr userdata
);

JavaScript Syntax

ChannelGroup.getUserData(
    userdata
    // writes value to userdata.val
);
Parameters

userdata
    Address of a variable to receive data that the user has stored 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.
See Also

- ChannelControl::setUserData
Firelight Technologies FMOD Studio API
ChannelGroup::getVolume

Retrieves the volume level.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getVolume(  
    float *volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetVolume(  
    FMOD_CHANNELGROUP *channelgroup,
    float *volume
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getVolume(  
    out float volume
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getVolume(  
    volume // writes value to volume.val
);
```
Parameters

volume
  Address of a variable to receive the linear volume level.
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

- ChannelControl::setVolume
Firelight Technologies FMOD Studio API
**ChannelGroup::getVolumeRamp**

Retrieves whether volume ramp is enabled.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::getVolumeRamp(
    bool *ramp
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_GetVolumeRamp(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_Bool *ramp
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.getVolumeRamp(
    out bool ramp
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.getVolumeRamp(  
    ramp  
    // writes value to ramp.val
);
```
Parameters

ramp
   Address of a variable to receive the volume ramp 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

- ChannelControl::setVolumeRamp
Firelight Technologies FMOD Studio API
ChannelGroup::isPlaying

Retrieves the playing state.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::isPlaying(
    bool *isplaying
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_IsPlaying(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL *isplaying
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.isPlaying(
    out bool isplaying
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.isPlaying(
    isplaying // writes value to isplaying.val
);
```
Parameters

isplaying
  Address of a variable that receives the current playing status, true = currently playing a sound, false = 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 Studio API
ChannelGroup::release

Frees a channel group.

C++ Syntax

FMOD_RESULT ChannelGroup::release();

C Syntax

FMOD_RESULT FMOD_ChannelGroup_Release(FMOD_CHANNELGROUP *channelgroup)

C# Syntax

RESULT ChannelGroup.release();

JavaScript Syntax

ChannelGroup.release();
Return Values

If the function succeeds then the return value 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 (and 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 Studio API
ChannelGroup::removeDSP

Remove a particular DSP unit from the DSP chain.

C++ Syntax

```c++
FMOD_RESULT ChannelGroup::removeDSP(
    DSP *dsp
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_RemoveDSP(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_DSP *dsp
);
```

C# Syntax

```csharp
RESULT ChannelGroup.removeDSP(
    DSP dsp
);
```

JavaScript Syntax

```javascript
ChannelGroup.removeDSP(
    dsp
);
```
Parameters

dsp

Pointer to a DSP unit (that exists in the DSP chain) you wish to remove.
Return Values

If the function succeeds then the return 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

- ChannelControl::addDSP
- ChannelControl::getDSP
- ChannelControl::getNumDSPs
Firelight Technologies FMOD Studio API
ChannelGroup::removeFadePoints

Remove volume fade points on the timeline. This function will remove multiple fade points with a single call if the points lay between the 2 specified clock values (inclusive).

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::removeFadePoints(
    unsigned long long dspclock_start,
    unsigned long long dspclock_end
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_RemoveFadePoints(
    FMOD_CHANNELGROUP *channelgroup,
    unsigned long long dspclock_start,
    unsigned long long dspclock_end
);
```

C# Syntax

```csharp
RESULT ChannelGroup.removeFadePoints(
    ulong dspclock_start,
    ulong dspclock_end
);
```

JavaScript Syntax

```javascript
ChannelGroup.removeFadePoints(
    dspclock_start,
    dspclock_end
);
```
Parameters

dspclock_start
   DSP clock of the parent channel group to start removing fade points from.
dspclock_end
   DSP clock of the parent channel group to start removing fade points 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.
See Also

- ChannelControl::addFadePoint
- ChannelControl::getFadePoints
Firelight Technologies FMOD Studio API
ChannelGroup::set3DAttributes

Sets the position and velocity used to apply panning, attenuation and doppler.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::set3DAttributes(
    const FMOD_VECTOR *pos,
    const FMOD_VECTOR *vel,
    const FMOD_VECTOR *alt_pan_pos
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_Set3DAttributes(
    FMOD_CHANNELGROUP *channelgroup,
    const FMOD_VECTOR *pos,
    const FMOD_VECTOR *vel,
    const FMOD_VECTOR *alt_pan_pos
);
```

C# Syntax

```csharp
RESULT ChannelGroup.set3DAttributes(
    ref VECTOR pos,
    ref VECTOR vel,
    ref VECTOR alt_pan_pos
);
```

JavaScript Syntax

```javascript
ChannelGroup.set3DAttributes(
    pos,
    vel,
    alt_pan_pos
);
```
**Parameters**

pos

Position in 3D space used for panning and attenuation. Optional, specify 0 or NULL to ignore.

vel

Velocity in 'distance units per second' (see remarks) in 3D space. Optional, specify 0 or NULL to ignore.

alt_pan_pos

(Unimplemented).
Return Values

If the function succeeds then the return value 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 ChannelControl::set3DSpread.

Vectors should use your chosen coordinate system, see 3D sounds for more information.
See Also

- ChannelControl::get3DAttributes
- ChannelControl::set3DSpread
- System::set3DSettings
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
ChannelGroup::set3DConeOrientation

Sets the orientation of the sound projection cone.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::set3DConeOrientation(
    FMOD_VECTOR *orientation
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Set3DConeOrientation(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_VECTOR *orientation
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.set3DConeOrientation(
    ref VECTOR orientation
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.set3DConeOrientation(
    orientation
);
```
Parameters

orientation
Coordinates of the sound cone orientation vector, 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.
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

- ChannelControl::get3DConeOrientation
- ChannelControl::set3DConeSettings
- Sound::set3DConeSettings
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
ChannelGroup::set3D Cone Settings

Sets the angles that define the sound projection cone including the volume when outside the cone.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::set3D Cone Settings(
    float insideconeangle,
    float outsideconeangle,
    float outsidevolume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Set3D Cone Settings(
    FMOD_CHANNELGROUP *channelgroup,
    float insideconeangle,
    float outsideconeangle,
    float outsidevolume
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.set3D Cone Settings(
    float insideconeangle,
    float outsideconeangle,
    float outsidevolume
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.set3D Cone Settings(
    insideconeangle,
    outsideconeangle,
    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 0.0 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

To define the parameters per sound use Sound::set3D ConeSettings.
See Also

- ChannelControl::get3DConeSettings
- ChannelControl::set3DConeOrientation
- Sound::set3DConeSettings
Firelight Technologies FMOD Studio API
**ChannelGroup::set3DCustomRolloff**

Sets a custom rolloff curve to define how audio will attenuate over distance. Must be used in conjunction with `FMOD_3D_CUSTOMROLLOFF` flag to be activated.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::set3DCustomRolloff(
    FMOD_VECTOR *points,
    int numpoints
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Set3DCustomRolloff(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_VECTOR *points,
    int numpoints
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.set3DCustomRolloff(
    ref VECTOR points,
    int numpoints
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.set3DCustomRolloff(
    points,
    numpoints
);
```
Parameters

points
   Array of **FMOD_VECTOR** structures where x = distance and y = volume from 0.0 to 1.0. z should be set to 0.

numpoints
   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.

Values set with ChannelControl::setMinMaxDistance are meaningless when FMOD_3D_CUSTOMROLLOFF is used, their values are ignored.

Here is an example of a custom array of points.

```c
static FMOD_VECTOR curve[3] =
{
    { 0.0f,  1.0f,  0.0f },
    { 2.0f,  0.2f,  0.0f },
    { 20.0f, 0.0f,  0.0f }
};
```

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.

To define the parameters per sound use Sound::set3DCustomRolloff.
See Also

- ChannelControl::get3DCustomRolloff
- ChannelControl::setMinMaxDistance
- Sound::set3DCustomRolloff
- FMOD_VECTOR
- FMOD_3D_CUSTOMROLLOFF
Firelight Technologies FMOD Studio API
ChannelGroup::set3DDistanceFilter

Control the behaviour of a 3D distance filter, whether to enable or disable it, and frequency characteristics.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::set3DDistanceFilter(  
    bool custom,  
    float customLevel,  
    float centerFreq  
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Set3DDistanceFilter(  
    FMOD_CHANNELGROUP *channelgroup,  
    FMOD_BOOL custom,  
    float customLevel,  
    float centerFreq  
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.set3DDistanceFilter(  
    bool custom,  
    float customLevel,  
    float centerFreq  
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.set3DDistanceFilter(  
    custom,  
    customLevel,  
    centerFreq  
);
```
Parameters

custom
   Specify true to disable FMOD distance rolloff calculation. Default = false.

customLevel
   Specify a attenuation factor manually here, where 1.0 = no attenuation and 0 = complete attenuation. Default = 1.0.

centerFreq
   Specify a center frequency in hz for the high-pass filter used to simulate distance attenuation, from 10.0 to 22050.0. Default = 1500.0.
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

- ChannelControl::get3DDistanceFilter
Firelight Technologies FMOD Studio API
ChannelGroup::set3DDopplerLevel

Sets the amount by which doppler is scaled.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::set3DDopplerLevel(
    float level
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Set3DDopplerLevel(
    FMOD_CHANNELGROUP *channelgroup,
    float level
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.set3DDopplerLevel(
    float level
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.set3DDopplerLevel(
    level
);
```
Parameters

level

Doppler scale from 0.0 (none), to 1.0 (normal) to 5.0 (exaggerated), 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

- `ChannelControl::get3DDopplerLevel`
Firelight Technologies FMOD Studio API
**ChannelGroup::set3DLevel**

Sets how much the 3D engine has an effect on the channel, versus that set by 2D panning functions.

**C++ Syntax**

```
FMOD_RESULT ChannelGroup::set3DLevel(
    float level
);
```

**C Syntax**

```
FMOD_RESULT FMOD_ChannelGroup_Set3DLevel(
    FMOD_CHANNELGROUP *channelgroup,
    float level
);
```

**C# Syntax**

```
RESULT ChannelGroup.set3DLevel(
    float level
);
```

**JavaScript Syntax**

```
ChannelGroup.set3DLevel(
    level
);
```
Parameters

level

3D pan level from 0.0 (attenuation is ignored and panning as set by 2D panning functions) to 1.0 (pan and attenuate according to 3D position), 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

Only affects sounds created **FMOD_3D**.

2D panning functions include `ChannelControl::setPan`, `ChannelControl::setMixLevelsOutput`, `ChannelControl::setMixLevelsInput`, `ChannelControl::setMixMatrix`, etc

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).
See Also

- ChannelControl::get3DLevel
- ChannelControl::setPan
- ChannelControl::setMixLevelsOutput
- ChannelControl::setMixLevelsInput
- ChannelControl::setMixMatrix
Firelight Technologies FMOD Studio API
ChannelGroup::set3DMinMaxDistance

Sets the minimum and maximum audible distance.

C++ Syntax

FMOD_RESULT ChannelGroup::set3DMinMaxDistance(
    float mindistance,
    float maxdistance
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_Set3DMinMaxDistance(
    FMOD_CHANNELGROUP *channelgroup,
    float mindistance,
    float maxdistance
);

C# Syntax

RESULT ChannelGroup.set3DMinMaxDistance(
    float mindistance,
    float maxdistance
);

JavaScript Syntax

ChannelGroup.set3DMinMaxDistance(
    mindistance,
    maxdistance
);
Parameters

mindistance
    Minimum volume distance in 'units' (see remarks), default = 1.0.
maxdistance
    Maximum volume distance in 'units' (see remarks), default = 10000.0.
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 the listener is in-between the minimum distance and the sound source the volume will be at its maximum. As the listener moves from the minimum distance to the maximum distance the sound will attenuate following the rolloff curve set. When outside the maximum distance the sound will no longer attenuate.

Minimum distance 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 small mindistance such as 0.1. This would cause it to attenuate quickly and dissapear 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 causing the volume to stay at its loudest until the listener was 100 meters away, then it would be hundreds of meters more before it would fade out.

Maximum distance 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 zero, 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.

To define the min and max distance per sound use Sound::set3DMinMaxDistance.

If FMOD_3D_CUSTOMROLLOFF is used, then these values are stored, but ignored in 3D processing.
See Also

- ChannelControl::get3DMinMaxDistance
- System::set3DSettings
- Sound::set3DMinMaxDistance
- FMOD_3D_CUSTOMROLLOFF
Firelight Technologies FMOD Studio API
ChannelGroup::set3DOcclusion

Sets the occlusion factors manually for when the FMOD geometry engine is not being used.

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
);
```

C# Syntax

```csharp
RESULT ChannelGroup.set3DOcclusion(
    float directocclusion,
    float reverbocclusion
);
```

JavaScript Syntax

```javascript
ChannelGroup.set3DOcclusion(
    directocclusion,
    reverbocclusion
);
```
**Parameters**

**directocclusion**

Occlusion factor for the direct path, from 0.0 (not occluded) to 1.0 (fully occluded), default = 0.0.

**reverbocclusion**

Occlusion factor for the reverb mix, from 0.0 (not occluded) to 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

Normally the volume is simply attenuated by the 'directocclusion' factor however if `FMOD_INIT_CHANNEL_LOWPASS` is specified frequency filtering will be used with a very small CPU hit.
See Also

- ChannelControl::get3DOcclusion
- FMOD_INIT_CHANNEL_LOWPASS
Firelight Technologies FMOD Studio API
ChannelGroup::set3DSpread

Sets the spread of a 3D sound in speaker space.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::set3DSpread(float angle);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Set3DSpread(FMOD_CHANNELGROUP *channelgroup, float angle);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.set3DSpread(float angle);
```

**JavaScript Syntax**

```javascript
ChannelGroup.set3DSpread(angle);
```
Parameters

angle
Speaker spread angle. 0 = all sound channels are located at the same speaker location and is 'mono'. 360 = all sound channels 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

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'.

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 channel located 15 degrees apart from each other in the speaker array.

Mono sounds are spread as if they were a stereo signal, i.e. the signal is split into 2. The power will remain the same as it spreads 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!
See Also

- ChannelControl::get3DSpread
Firelight Technologies FMOD Studio API
ChannelGroup::setCallback

Sets a callback to perform action for a specific event.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::setCallback(
    FMOD_CHANNELCONTROL_CALLBACK callback
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_SetCallback(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_CHANNELCONTROL_CALLBACK callback
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.setCallback(
    CHANNEL_CALLBACK callback
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.setCallback(
    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 driven by `System::update` and will only occur when this function is called. This has the main advantage of far less complication due to thread issues, and allows all FMOD commands, including loading sounds and playing new sounds from the callback. It also allows any type of sound to have an end callback, no matter what it is. The only disadvantage is that 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` in between your return type and function name.

```cpp
FMOD_RESULT F_CALLBACK mycallback(FMOD_CHANNELCONTROL *chanControl,
{
    if (controlType == FMOD_CHANNELCONTROL_TYPE_CHANNEL)
    {
        FMOD::Channel *channel = (FMOD::Channel *)chanControl;
        // Channel specific functions here...
    }
    else
    {
        FMOD::ChannelGroup *group = (FMOD::ChannelGroup *)chanControl;
        // ChannelGroup specific functions here...
    }

    // ChannelControl generic functions here...

    return FMOD_OK;
}
```
See Also

- System::update
- FMOD_CHANNELCONTROL_CALLBACK
- FMOD_CHANNELCONTROL_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
ChannelGroup::setDSPIndex

Moves the position in the DSP chain of a specified DSP unit.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::setDSPIndex(
    DSP *dsp,
    int index
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_SetDSPIndex(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_DSP *dsp,
    int index
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.setDSPIndex(
    DSP dsp,
    int index
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.setDSPIndex(
    dsp,
    index
);
```
Parameters

dsp
   Pointer to a DSP unit that exists in the DSP chain.
index
   Offset in the DSP chain to move the DSP to, see
   \texttt{FMOD\_CHANNELCONTROL\_DSP\_INDEX} for special named offsets.
Return Values

If the function succeeds then the return value 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 reordering DSP units inside a Channel or ChannelGroup so that processing can happen in the desired order.

You can verify the order of the DSP chain using iteration via `ChannelControl::getNumDSPs` and `ChannelControl::getDSP` or with the FMOD Profiler tool.
See Also

- ChannelControl::getDSPIndex
- ChannelControl::getNumDSPs
- ChannelControl::getDSP
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
ChannelGroup::setDelay

Sets a start (and/or stop) time relative to the parent channel group DSP clock, with sample accuracy.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setDelay(
    unsigned long long dspclock_start,
    unsigned long long dspclock_end,
    bool stopchannels
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetDelay(
    FMOD_CHANNELGROUP *channelgroup,
    unsigned long long dspclock_start,
    unsigned long long dspclock_end,
    FMOD_BOOL stopchannels
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setDelay(
    ulong dspclock_start,
    ulong dspclock_end,
    bool stopchannels
);
```

JavaScript Syntax

```javascript
ChannelGroup.setDelay(
    dspclock_start,
    dspclock_end,
    stopchannels
);
```
Parameters

dspclock_start
DSP clock of the parent channel group to audibly start playing sound at, a value of 0 indicates no delay.
dspclock_end
DSP clock of the parent channel group to audibly stop playing sound at, a value of 0 indicates no delay.
stopchannels
TRUE = stop according to ChannelControl::isPlaying. FALSE = remain 'active' and a new start delay could start playback again at a later 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

Every channel and channel group has its own DSP Clock. A channel or channel group can be delayed relatively against its parent, with sample accurate positioning. To delay a sound, use the 'parent' channel group DSP clock to reference against when passing values into this function.

If a parent channel group changes its pitch, the start and stop times will still be correct as the parent clock is rate adjusted by that pitch.
See Also

- ChannelControl::getDelay
- ChannelControl::getDSPClock
- ChannelControl::isPlaying
Firelight Technologies FMOD Studio API
ChannelGroup::setFadePointRamp

Add a short 64 sample volume ramp to the specified time in the future using fade points.

C++ Syntax

FMOD_RESULT ChannelGroup::setFadePointRamp(  
    unsigned long long dspclock,  
    float volume  
);

C Syntax

FMOD_RESULT FMOD_ChannelGroup_SetFadePointRamp(  
    FMOD_CHANNELGROUP *channelgroup,  
    unsigned long long dspclock,  
    float volume  
);

C# Syntax

RESULT ChannelGroup.setFadePointRamp(  
    ulong dspclock,  
    float volume  
);

JavaScript Syntax

ChannelGroup.setFadePointRamp(  
    dspclock,  
    volume  
);
Parameters

dspclock
   DSP clock of the parent channel group when the volume will be ramped to.
volume
   Volume level where 0 is silent and 1.0 is normal volume. Amplification is supported.
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 a helper function that automatically ramps from the current fade volume to the newly provided volume at a specified time. It will clear any fade points set after this time. Use in conjunction with ChannelControl::setDelay stop delay, to ramp down volume before it stops. The user would specify the same clock value for the fade ramp and stop delay. This can also be used as a way to provide sample accurate delayed volume changes without clicks.
See Also

- `ChannelControl::setDelay`
- `ChannelControl::removeFadePoints`
- `ChannelControl::addFadePoint`
- `ChannelControl::getFadePoints`
Firelight Technologies FMOD Studio API
ChannelGroup::setLowPassGain

Sets the gain of the dry signal when lowpass filtering is applied.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setLowPassGain(
    float gain
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetLowPassGain(
    FMOD_CHANNELGROUP *channelgroup, 
    float gain
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setLowPassGain(
    float gain
);
```

JavaScript Syntax

```javascript
ChannelGroup.setLowPassGain(
    gain
);
```
Parameters

gain
   Linear gain level, from 0 (silent, full filtering) to 1.0 (full volume, no filtering), 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

Requires the built in lowpass to be created with `FMOD_INIT_CHANNEL_LOWPASS` or `FMOD_INIT_CHANNEL_DISTANCEFILTER`. 
See Also

- ChannelControl::getLowPassGain
Firelight Technologies FMOD Studio API
ChannelGroup::setMixLevelsInput

Sets the incoming volume level for each channel of a multi-channel sound. This is a helper to avoid calling ChannelControl::setMixMatrix.

A multi-channel sound is a single sound that contains from 1 to 32 channels of sound data, in an interleaved fashion. If in the extreme case, a 32ch wave file was used, an array of 32 floating point numbers denoting their volume levels would be passed in to the levels parameter, and 32 for the numlevels parameter.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::setMixLevelsInput(
  float *levels,
  int numlevels
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_SetMixLevelsInput(
  FMOD_CHANNELGROUP *channelgroup,
  float *levels,
  int numlevels
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.setMixLevelsInput(
  float[] levels,
  int numlevels
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.setMixLevelsInput(
  levels,
  numlevels
);
```
**Parameters**

levels
   Array of volume levels for each incoming channel.
numlevels
   Number of levels in the array, from 0 to 32 inclusive.
Return Values

If the function succeeds then the return value 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 example use case for this function is if the sound file has multiple channels in it with different musical parts to it, but they are all in sync with each other. This function can be used to fade in and out different tracks of the sound or to solo/mute tracks within it.

NOTE: This function overwrites any pan/output mixlevel by overwriting the ChannelControl's matrix if it exists. It will create an NxN matrix where the output levels are the same as the input levels. If you wish to fold this down to a lower channel count mix rather than staying at the input channel count, either create a custom matrix instead and use ChannelControl::setMixMatrix, or add a new DSP after the fader, that has a different channel format (ie with ChannelControl::getDSP and DSP::setChannelFormat).

NOTE: Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.
See Also

- ChannelControl::setMixMatrix
- ChannelControl::getMixMatrix
- ChannelControl::getDSP
- DSP::setChannelFormat
- FMOD_CHANNELCONTROL_DSP_INDEX
Firelight Technologies FMOD Studio API
ChannelGroup::setMixLevelsOutput

Sets the speaker volume levels for each speaker individually, this is a helper to avoid calling ChannelControl::setMixMatrix.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setMixLevelsOutput(
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float surroundleft,
    float surroundright,
    float backleft,
    float backright
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetMixLevelsOutput(
    FMOD_CHANNELGROUP *channelgroup,
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float surroundleft,
    float surroundright,
    float backleft,
    float backright
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setMixLevelsOutput(
    float frontleft,
    float frontright,
    float center,
    float lfe,
    float surroundleft,
    float surroundright,
    float backleft,
    float backright,
```

float backright
);

**JavaScript Syntax**

ChannelGroup.setMixLevelsOutput(
    frontleft,
    frontright,
    center,
    lfe,
    surroundleft,
    surroundright,
    backleft,
    backright
);

Parameters

frontleft
   Volume level for the front left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).
frontright
   Volume level for the front right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).
center
   Volume level for the center speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).
lfe
   Volume level for the subwoofer speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).
surroundleft
   Volume level for the surround left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).
surroundright
   Volume level for the surround right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).
backleft
   Volume level for the back left speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).
backright
   Volume level for the back right speaker of a multichannel speaker setup, 0.0 (silent), 1.0 (normal volume).
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

NOTE: This function overwrites any pan/mixlevel by overwriting the ChannelControl's Matrix.

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion. 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 ChannelControl::setMixMatrix.
See Also

- ChannelControl::setMixMatrix
- ChannelControl::getMixMatrix
Firelight Technologies FMOD Studio API
ChannelGroup::setMixMatrix

Sets a 2D pan matrix that maps input channels (columns) to output speakers (rows).

C++ Syntax

```c++
FMOD_RESULT ChannelGroup::setMixMatrix(
    float *matrix,
    int outchannels,
    int inchannels,
    int matrixhop
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetMixMatrix(
    FMOD_CHANNELGROUP *channelgroup,
    float *matrix,
    int outchannels,
    int inchannels,
    int matrixhop
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setMixMatrix(
    float[] matrix,
    int outchannels,
    int inchannels,
    int inchannel_hop
);
```

JavaScript Syntax

```javascript
ChannelGroup.setMixMatrix(
    matrix,
    outchannels,
    inchannels,
    inchannel_hop
);
```
Parameters

matrix
  Address of a 2 dimensional array of volume levels in row-major order. Each row represents an output speaker, each column represents an input channel.

outchannels
  Number of output channels (rows) in the matrix being passed in, from 0 to FMOD_MAX_CHANNEL_WIDTH inclusive.

inchannels
  Number of input channels (columns) in the matrix being passed in, from 0 to FMOD_MAX_CHANNEL_WIDTH inclusive.

matrixhop
  The width (total number of columns) of the matrix. Optional. If this is 0, inchannels will be taken as the width of the matrix. Maximum of FMOD_MAX_CHANNEL_WIDTH.
Return Values

If the function succeeds then the return value 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 gain for input channel 's' to output channel 't' is matrix[t * matrixhop + s].

Levels can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

The matrix size will generally be the size of the number of channels in the current speaker mode. Use `System::getSoftwareFormat` to determine this.

If a matrix already exists then the matrix passed in will applied over the top of it. The input matrix can be smaller than the existing matrix.

A 'unit' matrix allows a signal to pass through unchanged. For example for a 5.1 matrix a unit matrix would look like this:

```
[ 1 0 0 0 0 0 ]
[ 0 1 0 0 0 0 ]
[ 0 0 1 0 0 0 ]
[ 0 0 0 1 0 0 ]
[ 0 0 0 0 1 0 ]
[ 0 0 0 0 0 1 ]
```
See Also

- `ChannelControl::getMixMatrix`
- `ChannelControl::setPan`
- `ChannelControl::setMixLevelsOutput`
- `ChannelControl::setMixLevelsInput`
- `System::getSoftwareFormat`
- `FMOD_MAX_CHANNEL_WIDTH`
Firelight Technologies FMOD Studio API
ChannelGroup::setMode

Changes some attributes for a channel or channelgroup based on the mode passed in.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setMode(
    FMOD_MODE mode
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetMode(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_MODE mode
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setMode(
    MODE mode
);
```

JavaScript Syntax

```javascript
ChannelGroup.setMode(
    mode
);
```
Parameters

mode
    Mode bits to set.
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

Flags supported:

- **FMOD_LOOP_OFF**
- **FMOD_LOOP_NORMAL**
- **FMOD_LOOP_BIDI**
- **FMOD_2D**
- **FMOD_3D**
- **FMOD_3D_HEADRELATIVE**
- **FMOD_3D_WORLDRELATIVE**
- **FMOD_3D_INVERSEROLLOFF**
- **FMOD_3D_LINEARROLLOFF**
- **FMOD_3D_LINESQUAREROLLOFF**
- **FMOD_3D_CUSTOMROLLOFF**
- **FMOD_3D_IGNOREGEOMETRY**
- **FMOD_VIRTUAL_PLAYFROMSTART**

Issues with streamed audio:

When changing the loop mode, sounds created with `System::createStream` or `FMOD_CREATESTREAM` may have already 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 by calling `Channel::setPosition`. Note this will usually only happen if you have sounds or loop points that are smaller than the stream decode buffer size.

Issues with PCM samples:

When changing the loop mode of sounds created with with `System::createSound` or `FMOD_CREATESAMPLE`, 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 (i.e. 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` or `FMOD_VIRTUAL_PLAYFROMSTART` is not specified, the flag will be cleared if it was specified previously.
See Also

- `Channel::getMode`
- `Channel::setPosition`
- `Sound::setMode`
- `System::createStream`
- `System::createSound`
- `System::setStreamBufferSize`
- `FMOD_CREATESOUNDEXINFO`
- `FMOD_MODE`
Firelight Technologies FMOD Studio API
ChannelGroup::setMute

Sets the mute state effectively silencing it or returning it to its normal volume.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setMute(
    bool mute
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetMute(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL mute
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setMute(
    bool mute
);
```

JavaScript Syntax

```javascript
ChannelGroup.setMute(
    mute
);
```
Parameters

mute
  Mute state, true = mute (silent), false = 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

Each channel and channel group has its own mute state, muting a channel group will mute all child channels but will not affect their individual setting. Calling `ChannelControl::getMute` will always return the value you set.
See Also

- ChannelControl::getMute
Firelight Technologies FMOD Studio API
ChannelGroup::setPan

Sets the pan level, this is a helper to avoid calling ChannelControl::setMixMatrix.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::setPan(
    float pan
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_SetPan(
    FMOD_CHANNELGROUP *channelgroup,
    float pan
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.setPan(
    float pan
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.setPan(
    pan
);
```
Parameters

pan
Pan level, from -1.0 (left) to 1.0 (right), default = 0 (center).
Return Values

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

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

- `ChannelControl::setMixMatrix`
- `ChannelControl::getMixMatrix`
Firelight Technologies FMOD Studio API
ChannelGroup::setPaused

Sets the paused state.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setPaused(
    bool paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetPaused(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL paused
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setPaused(
    bool paused
);
```

JavaScript Syntax

```javascript
ChannelGroup.setPaused(
    paused
);
```
Parameters

paused
    Paused state, 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

Each channel and channel group has its own paused state, pausing a channel group will pause all contained channels but will not affect their individual setting. Calling `ChannelControl::getPaused` will always return the value you set.
See Also

- `ChannelControl::getPaused`
Firelight Technologies FMOD Studio API
ChannelGroup::setPitch

Sets the pitch value.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setPitch(
    float pitch
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetPitch(
    FMOD_CHANNELGROUP *channelgroup,
    float pitch
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setPitch(
    float pitch
);
```

JavaScript Syntax

```javascript
ChannelGroup.setPitch(
    pitch
);
```
Parameters

pitch
   Pitch value, 0.5 = half pitch, 2.0 = double pitch, etc 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 scales existing frequency values by the pitch.
See Also

- ChannelControl::getPitch
- Channel::setFrequency
- Channel::getFrequency
Firelight Technologies FMOD Studio API
ChannelGroup::setReverbProperties

Sets the wet level (or send level) of a particular reverb instance.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::setReverbProperties(
    int instance,
    float wet
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_setReverbProperties(
    FMOD_CHANNELGROUP *channelgroup,
    int instance,
    float wet
);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.SetReverbProperties(
    int instance,
    float wet
);
```

**JavaScript Syntax**

```javascript
ChannelGroup.setReverbProperties(
    instance,
    wet
);
```
Parameters

instance
  Index of the particular reverb instance to target, from 0 to
  FMOD_REVERB_MAXINSTANCES inclusive.

wet
  Send level for the signal to the reverb, from 0 (none) to 1.0 (full), default =
  1.0 for Channels, 0.0 for ChannelGroups. 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

A Channel is automatically connected to all existing reverb instances due to the default wet level of 1.0. A ChannelGroup however will not send to any reverb by default requiring an explicit call to this function.

A ChannelGroup reverb is optimal for the case where you want to send 1 mixed signal to the reverb, rather than a lot of individual channel reverb sends. It is advisable to do this to reduce CPU if you have many Channels inside a ChannelGroup.

Keep in mind when setting a wet level for a ChannelGroup, any Channels under that ChannelGroup will still have their existing sends to the reverb. To avoid this doubling up you should explicitly set the Channel wet levels to 0.0.
See Also

- ChannelControl::getReverbProperties
Firelight Technologies FMOD Studio API
ChannelGroup::setUserData

Sets a user value that can be retrieved with ChannelControl::getUserData.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setUserData(
    void *userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetUserData(
    FMOD_CHANNELGROUP *channelgroup,
    void *userdata
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setUserData(
    IntPtr userdata
);
```

JavaScript Syntax

```javascript
ChannelGroup.setUserData(
    userdata
);
```
Parameters

userdata
   Data that the user wishes stored 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

You can use this to store a pointer to any wrapper or internal class that is associated with this object. This is especially useful for callbacks where you need to get back access to your own objects.
See Also

- [ChannelControl::getUserData](#)
Firelight Technologies FMOD Studio API
ChannelGroup::setVolume

Sets the volume level linearly.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setVolume(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetVolume(
    FMOD_CHANNELGROUP *channelgroup,
    float volume
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setVolume(
    float volume
);
```

JavaScript Syntax

```javascript
ChannelGroup.setVolume(
    volume
);
```
Parameters

volume
   Linear volume level, 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

Volume level can be below 0 to invert a signal and above 1 to amplify the signal. Note that increasing the signal level too far may cause audible distortion.

Sound::setDefault can be used to change the default volume for any channels played using that sound.
See Also

- ChannelControl::getVolume
- Sound::setDefaults
Firelight Technologies FMOD Studio API
ChannelGroup::setVolumeRamp

Sets whether the channel automatically ramps when setting volumes.

C++ Syntax

```cpp
FMOD_RESULT ChannelGroup::setVolumeRamp(
    bool ramp
);
```

C Syntax

```c
FMOD_RESULT FMOD_ChannelGroup_SetVolumeRamp(
    FMOD_CHANNELGROUP *channelgroup,
    FMOD_BOOL ramp
);
```

C# Syntax

```csharp
RESULT ChannelGroup.setVolumeRamp(
    bool ramp
);
```

JavaScript Syntax

```javascript
ChannelGroup.setVolumeRamp(
    ramp
);
```
Parameters

ramp
  Whether to enable volume ramping.
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

When changing volumes on a non-paused channel, FMOD normally adds a small ramp to avoid a pop sound. This function allows that setting to be overridden and volume changes to be applied immediately.
See Also

- ChannelControl::getVolumeRamp
Firelight Technologies FMOD Studio API
ChannelGroup::stop

Stops the channel (or all channels in the channel group) from playing. Makes it available for re-use by the priority system.

**C++ Syntax**

```cpp
FMOD_RESULT ChannelGroup::stop();
```

**C Syntax**

```c
FMOD_RESULT FMOD_ChannelGroup_Stop(FMOD_CHANNELGROUP *channelgroup);
```

**C# Syntax**

```csharp
RESULT ChannelGroup.stop();
```

**JavaScript Syntax**

```javascript
ChannelGroup.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.
See Also

- System::playSound
Firelight Technologies FMOD Studio API
SoundGroup
Functions

SoundGroup::getMaxAudible  SoundGroup::getMaxAudibleBehavior
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 Studio API
**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_SOUNDGROUP *soundgroup,  
    int *maxaudible
);
```

**C# Syntax**

```csharp
RESULT SoundGroup.getMaxAudible(  
    out int maxaudible
);
```

**JavaScript Syntax**

```javascript
SoundGroup.getMaxAudible(  
    maxaudible  // writes value to maxaudible.val
);
```
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 Studio API
SoundGroup::getMaxAudibleBehavior

Retrieves the current max audible behavior method.

C++ Syntax

FMOD_RESULT SoundGroup::getMaxAudibleBehavior(
   FMOD_SOUNDGROUP_BEHAVIOR *behavior
);

C Syntax

FMOD_RESULT FMOD_SoundGroup_GetMaxAudibleBehavior(
   FMOD_SOUNDGROUP *soundgroup,
   FMOD_SOUNDGROUP_BEHAVIOR *behavior
);

C# Syntax

RESULT SoundGroup.getMaxAudibleBehavior(
   out SOUNDGROUP_BEHAVIOR behavior
);

JavaScript Syntax

SoundGroup.getMaxAudibleBehavior(  
   behavior // writes value to behavior.val
);
Parameters

behavior
   Address of a variable to receive the current sound group max playbacks behavior. Default is `FMOD_SOUNDP膨GROUP_BEA膨OR_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_SOUNDDGROUP_BEHAVIOR`
- `SoundGroup::setMaxAudibleBehavior`
- `SoundGroup::setMaxAudible`
- `SoundGroup::getMaxAudible`
- `SoundGroup::setMuteFadeSpeed`
- `SoundGroup::getMuteFadeSpeed`
- `System::createSoundGroup`
- `System::getMasterSoundGroup`
Firelight Technologies FMOD Studio API
SoundGroup::getMuteFadeSpeed

Retrieves the current time in seconds for FMOD_SOUNDDGROUP_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
);
```

C# Syntax

```csharp
RESULT SoundGroup.getMuteFadeSpeed(
  out float speed
);
```

JavaScript Syntax

```javascript
SoundGroup.getMuteFadeSpeed(
  speed // writes value to speed.val
);
```
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_SOUNDMODE_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 Studio API
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,      // writes value to name.val
    int namelen
);
```

C# Syntax

```csharp
RESULT SoundGroup.getName(
    StringBuilder name,      // writes value to name.val
    int namelen
);
```

JavaScript Syntax

```javascript
SoundGroup.getName(
    name
);
```
**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.
Remarks

JavaScript only:

Note: For the "name" parameter, the maximum string length is 512.
See Also

- System::createSoundGroup
- System::getMasterSoundGroup

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
**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_SOUNDGROUP *soundgroup,
    int *numplaying
);
```

**C# Syntax**

```csharp
RESULT SoundGroup.getNumPlaying(
    out int numplaying
);
```

**JavaScript Syntax**

```javascript
SoundGroup.getNumPlaying(
    numplaying // writes value to numplaying.val
);
```
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 Studio API
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
);
```

C# Syntax

```csharp
RESULT SoundGroup.getNumSounds(  
    out int numsounds
);
```

JavaScript Syntax

```javascript
SoundGroup.getNumSounds(  
    numsounds // writes value to numsounds.val
);
```
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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT SoundGroup.getSound(
    int index,
    out Sound sound
);
```

**JavaScript Syntax**

```javascript
SoundGroup.getSound(
    index,
    sound // writes value to sound.val
);
```
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 Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT SoundGroup.getSystemObject(
    out System system
);
```

**JavaScript Syntax**

```javascript
SoundGroup.getSystemObject(
    system
) // writes value to system.val
```
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 Studio API
**SoundGroup::getUserData**

Retrieves the user value 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);
```

**C# Syntax**

```csharp
RESULT SoundGroup.getUserData(
    out IntPtr userdata);
```

**JavaScript Syntax**

```javascript
SoundGroup.getUserData(
    userdata); // writes value to userdata.val
```
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 Studio API
SoundGroup::getVolume

Retrieves the volume for the sounds within a soundgroup.

C++ Syntax

FMOD_RESULT SoundGroup::getVolume(
    float *volume
);

C Syntax

FMOD_RESULT FMOD_SoundGroup_GetVolume(
    FMOD_SOUNDGROUP *soundgroup,
    float *volume
);

C# Syntax

RESULT SoundGroup.getVolume(
    out float volume
);

JavaScript Syntax

SoundGroup.getVolume(
    volume
); // writes value to volume.val
Parameters

volume
Address of a variable to receive the soundgroup volume level, 0.0 = silent, 1.0 = full volume. Default = 1.0. Negative volumes and amplification (> 1.0) are supported.
Return Values

If the function succeeds then the return 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 Studio API
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);
```

**C# Syntax**

```csharp
RESULT SoundGroup.release();
```

**JavaScript Syntax**

```javascript
SoundGroup.release();
```
**Return Values**

If the function succeeds then the return value 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 Studio API
**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_SOUNDGROUP *soundgroup,
    int maxaudible
);
```

**C# Syntax**

```csharp
RESULT SoundGroup.setMaxAudible(
    int maxaudible
);
```

**JavaScript Syntax**

```javascript
SoundGroup.setMaxAudible(
    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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT SoundGroup.setMaxAudibleBehavior(  
    SOUNDGROUP_BEHAVIOR behavior
);
```

**JavaScript Syntax**

```javascript
SoundGroup.setMaxAudibleBehavior(  
    behavior
);
```
Parameters

behavior
   Specify a behavior determined with a
   FMOD_SOUNDDGROUP_BEHAVIOR flag. Default is
   FMOD_SOUNDDGROUP_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_SOUNDCOMPONENT_BEHAVIOR**
- **SoundGroup::getMaxAudibleBehavior**
- **SoundGroup::setMaxAudible**
- **SoundGroup::getMaxAudible**
- **SoundGroup::setMuteFadeSpeed**
- **SoundGroup::getMuteFadeSpeed**
- **System::createSoundGroup**
- **System::getMasterSoundGroup**
Firelight Technologies FMOD Studio API
SoundGroup::setMuteFadeSpeed

Specify a time in seconds for FMOD_SOUNDDGROUP_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_SOUNDDGROUP_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

FMOD_RESULT SoundGroup::setMuteFadeSpeed(
    float speed
);

C Syntax

FMOD_RESULT FMOD_SoundGroup_SetMuteFadeSpeed(
    FMOD_SOUNDDGROUP *soundgroup,
    float speed
);

C# Syntax

RESULT SoundGroup.setMuteFadeSpeed(
    float speed
);

JavaScript Syntax

SoundGroup.setMuteFadeSpeed(
    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 \texttt{FMOD\_SOUNDBLOCK\_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 Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT SoundGroup.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
SoundGroup.setUserData(
    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 `SoundGroup::getUserData` would help in the identification of the object.
See Also

- SoundGroup::getUserData
- System::createSoundGroup
- System::getMasterSoundGroup
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT SoundGroup.setVolume(
    float volume
);
```

**JavaScript Syntax**

```javascript
SoundGroup.setVolume(
    volume
);
```
Parameters

volume
A linear volume level. 0.0 = silent, 1.0 = full volume. Default = 1.0. Negative volumes and amplification (> 1.0) are supported.
Return Values

If the function succeeds then the return 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 Studio API
**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);
```

**C# Syntax**

```csharp
RESULT SoundGroup.stop();
```

**JavaScript Syntax**

```javascript
SoundGroup.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.
See Also

- System::playSound
- System::createSoundGroup
- System::getMasterSoundGroup
Firelight Technologies FMOD Studio API
DSP
Functions

DSP::addInput  DSP::disconnectAll
DSP::disconnectFrom
DSP::getActive
DSP::getBypass
DSP::getChannelFormat
DSP::getDataParameterIndex
DSP::getIdle
DSP::getInfo
DSP::getInput
DSP::getMeteringEnabled
DSP::getMeteringInfo
DSP::getNumInputs
DSP::getNumOutputs
DSP::getNumParameters
DSP::getOutput
DSP::getOutputChannelFormat
DSP::getParameterBool
DSP::getParameterData
DSP::getParameterFloat
DSP::getParameterInfo
DSP::getParameterInt
DSP::getSystemObject
DSP::getType
DSP::getUserData
DSP::getWetDryMix
DSP::release
DSP::reset
DSP::setActive
DSP::setBypass
DSP::setChannelFormat
DSP::setMeteringEnabled
DSP::setParameterBool
DSP::setParameterData
DSP::setParameterFloat
DSP::setParameterInt
DSP::setUserData
DSP::setWetDryMix
DSP::showConfigDialog
DSP::addInput

Adds the specified DSP unit as an input of the DSP object.

C++ Syntax

```cpp
FMOD_RESULT DSP::addInput(
    FMOD::DSP *input,
    FMOD::DSPConnection **connection,
    FMOD_DSPCONNECTION_TYPE type
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_AddInput(
    FMOD_DSP *dsp,
    FMOD_DSP *input,
    FMOD_DSPCONNECTION **connection,
    FMOD_DSPCONNECTION_TYPE type
);
```

C# Syntax

```csharp
RESULT DSP.addInput(
    DSP target,
    out DSPConnection connection,
    DSPCONNECTION_TYPE type
);
```

JavaScript Syntax

```javascript
DSP.addInput(
    input,
    connection, // writes value to connection.val
    type
);
```
Parameters

input
The DSP unit to add as an input of the current unit.

collection
The connection between the 2 units. Optional. Specify 0 or NULL to ignore.

type
The type of connection between the 2 units. See
FMOD_DSPCONNECTION_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

If you want to add a unit as an output of another unit, then add 'this' unit as an input of that unit instead.

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.
See Also

- DSP::getNumInputs
- DSP::getInput
- DSP::getNumOutputs
- DSP::disconnectFrom
- FMOD_DSPCONNECTION_TYPE

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT DSP.disconnectAll(
    bool inputs,
    bool outputs
);
```

JavaScript Syntax

```javascript
DSP.disconnectAll(
    inputs,
    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 Studio API
DSP::disconnectFrom

Disconnect the DSP unit from the specified input.

C++ Syntax

```cpp
FMOD_RESULT DSP::disconnectFrom(
    FMOD::DSP *target,
    FMOD::DSPConnection *connection
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_DisconnectFrom(
    FMOD_DSP *dsp,
    FMOD_DSP *target,
    FMOD_DSPCONNECTION *connection
);
```

C# Syntax

```csharp
RESULT DSP.disconnectFrom(
    DSP target,
    DSPConnection connection
);
```

JavaScript Syntax

```javascript
DSP.disconnectFrom(
    target,
    connection
);
```
Parameters

target
   The input unit that this unit is to be disconnected from. Specify 0 or NULL to disconnect the unit from all outputs and inputs.

connection
   If there is more than one connection between 2 dsp units, this can be used to define which of the connections should be disconnected.
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

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 Studio API
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
);
```

C# Syntax

```csharp
RESULT DSP.getActive(
    out bool active
);
```

JavaScript Syntax

```javascript
DSP.getActive(
    active   // writes value to active.val
);
```
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 Studio API
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
);
```

C# Syntax

```csharp
RESULT DSP.getBypass(
    out bool bypass
);
```

JavaScript Syntax

```javascript
DSP.getBypass(
    bypass
); // writes value to bypass.val
```
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 \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

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 Studio API
**DSP::getChannelFormat**

Gets the input signal format for a dsp units read/process callback, to determine which speakers the signal will be processed on and how many channels will be processed.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::getChannelFormat(
    FMOD_CHANNELMASK *channelmask,
    int *numchannels,
    FMOD_SPEAKERMODE *source_speakermode
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_GetChannelFormat(
    FMOD_DSP *dsp,
    FMOD_CHANNELMASK *channelmask,
    int *numchannels,
    FMOD_SPEAKERMODE *source_speakermode
);
```

**C# Syntax**

```csharp
RESULT DSP.getChannelFormat(
    out CHANNELMASK channelmask,
    out int numchannels,
    out SPEAKERMODE source_speakermode
);
```

**JavaScript Syntax**

```javascript
DSP.getChannelFormat(  
    channelmask,  // writes value to channelmask.va
    numchannels,  // writes value to numchannels.va
    source_speakermode  // writes value to source_speaker
);
```
Parameters

channelmask
Address of a variable that receives the `FMOD_CHANNELMASK` which determines which speakers are represented by the channels in the input signal.

numchannels
Address of a variable that receives the number of channels to be processed on this unit.

source_speakermode
Address of a variable that receives the source speaker mode where the signal came 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

source_speakermode is informational, when channelmask describes what bits are active, and numchannels describes how many channels are in a buffer, source_speakermode describes where the channels originated from. For example if numchannels = 2 then this could describe for the DSP if the original signal started from a stereo signal or a 5.1 signal. In the 5.1 signal the channels described might only represent 2 surround speakers for example.
See Also

- FMOD_CHANNELMASK
- DSP::setChannelFormat
Firelight Technologies FMOD Studio API
DSP::getDataParameterIndex

Retrieve the index of the first data parameter of a particular data type.

C++ Syntax

FMOD_RESULT DSP::getDataParameterIndex(
    int datatype,
    int *index
);

C Syntax

FMOD_RESULT FMOD_DSP_GetDataParameterIndex(
    FMOD_DSP *dsp,
    int datatype,
    int *index
);

C# Syntax

RESULT DSP.getDataParameterIndex(
    int datatype,
    out int index
);

JavaScript Syntax

DSP.getDataParameterIndex(
    datatype,
    index // writes value to index.val
);
**Parameters**

**datatype**
- The type of data to find. This would usually be set to a value defined in `FMOD_DSP_PARAMETER_DATA_TYPE` but can be any value for custom types.

**index**
- Contains the index of the first data parameter of type 'datatype' after the function is called. Will be -1 if no matches were found. Can be 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 `FMOD_RESULT` enumeration.
Remarks

This function returns `FMOD_OK` if a parameter of matching type is found and `FMOD_ERR_INVALID_PARAM` if no matches were found.

The return code can therefore be used to check whether the DSP supports specific functionality through data parameters of certain types without the need to pass in 'index'.
See Also

- FMOD_DSP_PARAMETER_DATA_TYPE
Firelight Technologies FMOD Studio API
DSP::getIdle

Retrieves the idle state of a DSP. A DSP is idle when no signal is coming into it. This can be a useful method of determining if a DSP sub branch is finished processing, so it can be disconnected for example.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::getIdle(
    bool *idle
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_GetIdle(
    FMOD_DSP *dsp,
    FMOD_BOOL *idle
);
```

**C# Syntax**

```csharp
RESULT DSP.getIdle(
    out bool idle
);
```

**JavaScript Syntax**

```javascript
DSP.getIdle(
    idle // writes value to idle.val
);
```
Parameters

idle
Address of a variable to receive the idle state for the DSP.
Return Values

If the function succeeds then the return value 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 idle state takes into account things like tails of echo filters, even if a wavetable or dsp has finished generating sound. When all nodes in a graph have finished processing, only then will it set the top level DSP state to idle.
Firelight Technologies FMOD Studio API
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

FMOD_RESULT DSP::getInfo(
    char *name,
    unsigned int *version,
    int *channels,
    int *configwidth,
    int *configheight
);

C Syntax

FMOD_RESULT FMOD_DSP_GetInfo(
    FMOD_DSP *dsp,
    char *name,
    unsigned int *version,
    int *channels,
    int *configwidth,
    int *configheight
);

C# Syntax

RESULT DSP.getInfo(
    StringBuilder name,
    out uint version,
    out int channels,
    out int configwidth,
    out int configheight
);

JavaScript Syntax

DSP.getInfo(
    name, // writes value to name.val
    version, // writes value to version.val
channels,
configwidth,
configheight
);

// writes value to channels.val
// writes value to configwidth.val
// writes value to configheight.val
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 BBBB 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.
Remarks

JavaScript only:

Note: For the "name" parameter, the maximum string length is 512.
See Also

- DSP::showConfigDialog
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT DSP.getInput(
    int index,
    out DSP input,
    out DSPConnection inputconnection
);
```

**JavaScript Syntax**

```javascript
DSP.getInput(
    index,
    input, // writes value to input.val
    inputconnection // writes value to 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 Studio API
DSP::getMeteringEnabled

Retrieve the information about metering for a particular DSP to see if it is enabled or not.

C++ Syntax

```cpp
FMOD_RESULT DSP::getMeteringEnabled(
    bool *inputEnabled,
    bool *outputEnabled
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetMeteringEnabled(
    FMOD_DSP *dsp,
    FMOD_BOOL *inputEnabled,
    FMOD_BOOL *outputEnabled
);
```

C# Syntax

```csharp
RESULT DSP.getMeteringEnabled(
    out bool inputEnabled,
    out bool outputEnabled
);
```

JavaScript Syntax

```javascript
DSP.getMeteringEnabled(
    inputEnabled, // writes value to inputEnabled.v
    outputEnabled // writes value to outputEnabled.
);
```
Parameters

inputEnabled
Address of a variable to receive the metering enabled state for the DSP, for the input signal (pre-processing). true = on, false = off. Optional. Specify 0 or NULL to ignore.

outputEnabled
Address of a variable to receive the metering enabled state for the DSP, for the output signal (post-processing). true = on, false = off. 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

`FMOD_INIT_PROFILE_METER_ALL` with `System::init` will automatically turn on metering for all DSP units inside the FMOD mixer graph.
See Also

- DSP::setMeteringEnabled
- DSP::getMeteringInfo
- System::init
Firelight Technologies FMOD Studio API
DSP::getMeteringInfo

Retrieve the metering information for a particular DSP.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::getMeteringInfo(
    FMOD_DSP_METERING_INFO *inputInfo,
    FMOD_DSP_METERING_INFO *outputInfo
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_GetMeteringInfo(
    FMOD_DSP *dsp,
    FMOD_DSP_METERING_INFO *inputInfo,
    FMOD_DSP_METERING_INFO *outputInfo
);
```

**C# Syntax**

```csharp
RESULT DSP.getMeteringInfo(
    DSP_METERING_INFO inputInfo,
    DSP_METERING_INFO outputInfo
);
```

**JavaScript Syntax**

```javascript
DSP.getMeteringInfo(
    inputInfo, // writes value to inputInfo.val
    outputInfo // writes value to outputInfo.val
);
```
**Parameters**

**inputInfo**
Address of a variable to receive metering information for the DSP, for the input signal (pre-processing). true = on, false = off. Optional. Specify 0 or NULL to ignore.

**outputInfo**
Address of a variable to receive metering information for the DSP, for the output signal (post-processing). true = on, false = off. 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

**FMOD_INIT_PROFILE_METER_ALL** with **System::init** will automatically turn on metering for all DSP units inside the FMOD mixer graph.
See Also

- FMOD_DSP_METERING_INFO
- DSP::setMeteringEnabled
- DSP::getMeteringEnabled
- System::init
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT DSP.getNumInputs(
    out int numinputs
);
```

**JavaScript Syntax**

```javascript
DSP.getNumInputs(  
    numinputs      // writes value to numinputs.val
 );
```
**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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT DSP.getNumOutputs(
    out int numoutputs
);
```

**JavaScript Syntax**

```javascript
DSP.getNumOutputs(
    numoutputs // writes value to numoutputs.val
);
```
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 Studio API
DSP::getNumParameters

Retrieves the number of parameters a DSP unit has to control its behaviour.

C++ Syntax

FMOD_RESULT DSP::getNumParameters(
   int *numparams
);

C Syntax

FMOD_RESULT FMOD_DSP_GetNumParameters(
   FMOD_DSP *dsp,
   int *numparams
);

C# Syntax

RESULT DSP.getNumParameters(
   out int numparams
);

JavaScript Syntax

DSP.getNumParameters(
   numparams
); // writes value to numparams.val
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::getParameterInfo`. 
See Also

- DSP::getParameterInfo
- DSP::setParameterFloat
- DSP::setParameterInt
- DSP::setParameterBool
- DSP::setParameterData
- DSP::getParameterFloat
- DSP::getParameterInt
- DSP::getParameterBool
- DSP::getParameterData
Firelight Technologies FMOD Studio API
DSP::getOutput

Retrieves a pointer to a DSP unit which is acting as an output to this unit.

C++ Syntax

FMOD_RESULT DSP::getOutput(
    int index,
    FMOD::DSP **output,
    FMOD::DSPConnection **outputconnection
);

C Syntax

FMOD_RESULT FMOD_DSP_GetOutput(
    FMOD_DSP *dsp,
    int index,
    FMOD_DSP **output,
    FMOD_DSPCONNECTION **outputconnection
);

C# Syntax

RESULT DSP.getOutput(
    int index,
    out DSP output,
    out DSPConnection outputconnection
);

JavaScript Syntax

DSP.getOutput(
    index,
    output, // writes value to output.val
    outputconnection // writes value to 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 Studio API
DSP::getOutputChannelFormat

Call the DSP process function to retrieve the output signal format for a DSP based on input values.

C++ Syntax

```cpp
FMOD_RESULT DSP::getOutputChannelFormat(  
    FMOD_CHANNELMASK inmask,  
    int inchannels,  
    FMOD_SPEAKERMODE inspeakermode,  
    FMOD_CHANNELMASK *outmask,  
    int *outchannels,  
    FMOD_SPEAKERMODE *outspeakermode
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetOutputChannelFormat(  
    FMOD_DSP *dsp,  
    FMOD_CHANNELMASK inmask,  
    int inchannels,  
    FMOD_SPEAKERMODE inspeakermode,  
    FMOD_CHANNELMASK *outmask,  
    int *outchannels,  
    FMOD_SPEAKERMODE *outspeakermode
);
```

C# Syntax

```csharp
RESULT DSP.getOutputChannelFormat(  
    CHANNELMASK inmask,  
    int inchannels,  
    SPEAKERMODE inspeakermode,  
    out CHANNELMASK outmask,  
    out int outchannels,  
    out SPEAKERMODE outspeakermode
);
```

JavaScript Syntax
DSP.getOutputChannelFormat(
    inmask,
    inchannels,
    inspeakermode,
    outmask,       // writes value to outmask.val
    outchannels,   // writes value to outchannels.val
    outspeakermode // writes value to outspeakermode
);
Parameters

**inmask**
Channel bitmask representing the speakers enabled for the incoming signal. For example a 5.1 signal could have inchannels 2 that represent

```
FMOD_CHANNELMASK_SURROUND_LEFT
FMOD_CHANNELMASK_SURROUND_RIGHT
```

**inchannels**
Number of channels for the incoming signal.

**inspeckermode**
Speaker mode for the incoming signal.

**outmask**
Address of a variable to receive the DSP unit's output mask, based on the DSP units preference and settings.

**outchannels**
Address of a variable to receive the DSP unit's output channel count, based on the DSP units preference and settings.

**outspeckermode**
Address of a variable to receive the DSP unit's output speaker mode, based on the DSP units preference and 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.
Remarks

A DSP unit may be an up mixer or down mixer for example. In this case if you specified 6 in for a downmixer, it may provide you with 2 out for example. Generally the input values will be reproduced for the output values, but some DSP units will want to alter the output format.
See Also

- DSP::getChannelFormat
Firelight Technologies FMOD Studio API
DSP::getParameterBool

Retrieves a DSP unit's boolean parameter by index. To find out the parameter names and range, see the see also field.

C++ Syntax

FMOD_RESULT DSP::getParameterBool(
    int index,
    bool *value,
    char *valuestr,
    int valuestrlen
);

C Syntax

FMOD_RESULT FMOD_DSP_GetParameterBool(
    FMOD_DSP *dsp,
    int index,
    FMOD_BOOL *value,
    char *valuestr,
    int valuestrlen
);

C# Syntax

RESULT DSP.getParameterBool(
    int index,
    out bool value
);

JavaScript Syntax

DSP.getParameterBool(
    index,
    value, // writes value to value.val
    valuestr // writes value to valuestr.val
);
**Parameters**

**index**
Parameter index for this unit. Find the number of parameters with `DSP::getNumParameters`.

**value**
Address of a variable that receives the boolean parameter value for the parameter specified.

**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. Optional. Specify 0 or NULL to ignore.

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

The parameter properties can be retrieved with `DSP::getParameterInfo`.

**JavaScript only:**

Note: For the "valuestr" parameter, the maximum string length is 512.
See Also

- DSP::getParameterInfo
- DSP::getNumParameters
- DSP::setParameterBool
- FMOD_DSP_OSCILLATOR
- FMOD_DSP_LOWPASS
- FMOD_DSP_ITLOWPASS
- FMOD_DSP_HIGHPASS
- FMOD_DSP_ECHO
- FMOD_DSP_FLANGE
- FMOD_DSP_DISTORTION
- FMOD_DSP_NORMALIZE
- FMOD_DSP_LIMITER
- FMOD_DSP_PAREMEQ
- FMOD_DSP_PITCHSHIFT
- FMOD_DSP_CHORUS
- FMOD_DSP_ITECHO
- FMOD_DSP_COMPRESSOR
- FMOD_DSP_SFXREVERB
- FMOD_DSP_LOWPASS_SIMPLE
- FMOD_DSP_DELAY
- FMOD_DSP_TREMOLO
- FMOD_DSP_SEND
- FMOD_DSP_RETURN
- FMOD_DSP_HIGHPASS_SIMPLE
- FMOD_DSP_PAN
- FMOD_DSP_THREE_EQ
- FMOD_DSP_FFT
Firelight Technologies FMOD Studio API
DSP::getParameterData

Retrieves a DSP unit's data block parameter by index. To find out the parameter names and range, see the see also field.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::getParameterData(
    int index,
    void **data,
    unsigned int *length,
    char *valuestr,
    int valuestrlen
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_GetParameterData(
    FMOD_DSP *dsp,
    int index,
    void **data,
    unsigned int *length,
    char *valuestr,
    int valuestrlen
);
```

**C# Syntax**

```csharp
RESULT DSP.getParameterData(
    int index,
    out IntPtr data,
    out uint length
);
```

**JavaScript Syntax**

```javascript
DSP.getParameterData(
    index, // writes value to data.val
    data,  // writes value to length.val
    length, // writes value to valuestr.val
    valuestr
);
```
});
### Parameters

**index**
Parameter index for this unit. Find the number of parameters with `DSP::getNumParameters`.

**data**
Address of a variable that receives binary data for the parameter specified.

**length**
Address of a variable that receives the length of data block in bytes. Optional.

**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. Optional. Specify 0 or NULL to ignore.

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

The parameter properties can be retrieved with `DSP::getParameterInfo`.

**JavaScript only:**

Note: For the "valuestr" parameter, the maximum string length is 512.
See Also

- `DSP::getParameterInfo`
- `DSP::getNumParameters`
- `DSP::setParameterData`
- `FMOD_DSP_PARAMETER_DATA_TYPE`
- `FMOD_DSP_OSCILLATOR`
- `FMOD_DSP_LOWPASS`
- `FMOD_DSP_ITLOWPASS`
- `FMOD_DSP_HIGHPASS`
- `FMOD_DSP_ECHO`
- `FMOD_DSP_FLANGE`
- `FMOD_DSP_DISTORTION`
- `FMOD_DSP_NORMALIZE`
- `FMOD_DSP_LIMITER`
- `FMOD_DSP_PARAMEQ`
- `FMOD_DSP_PITCHSHIFT`
- `FMOD_DSP_CHORUS`
- `FMOD_DSP_ITECHO`
- `FMOD_DSP_COMPRESSOR`
- `FMOD_DSP_SFXREVERB`
- `FMOD_DSP_LOWPASS_SIMPLE`
- `FMOD_DSP_DELAY`
- `FMOD_DSP_TREMOLO`
- `FMOD_DSP_SEND`
- `FMOD_DSP_RETURN`
- `FMOD_DSP_HIGHPASS_SIMPLE`
- `FMOD_DSP_PAN`
- `FMOD_DSP_THREE_EQ`
- `FMOD_DSP_FFT`
Firelight Technologies FMOD Studio API
DSP::getParameterFloat

Retrieves a DSP unit's floating point parameter by index. To find out the parameter names and range, see the see also field.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::getParameterFloat(
    int index,
    float *value,
    char *valuestr,
    int valuestrlen
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_GetParameterFloat(
    FMOD_DSP *dsp,
    int index,
    float *value,
    char *valuestr,
    int valuestrlen
);
```

**C# Syntax**

```csharp
RESULT DSP.getParameterFloat(
    int index,
    out float value
);
```

**JavaScript Syntax**

```javascript
DSP.getParameterFloat(
    index,
    value, // writes value to value.val
    valuestr // writes value to valuestr.val
);
```
Parameters

index
Parameter index for this unit. Find the number of parameters with **DSP::getNumParameters**.

value
Address of a variable that receives the floating point parameter value for the parameter specified.

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. Optional. Specify 0 or NULL to ignore.

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.
Remarks

The parameter properties can be retrieved with DSP::getParameterInfo.

JavaScript only:

Note: For the "valuestr" parameter, the maximum string length is 512.
See Also

- DSP::getParameterInfo
- DSP::getNumParameters
- DSP::setParameterFloat
- FMOD_DSP_OSCILLATOR
- FMOD_DSP_LOWPASS
- FMOD_DSP_ITLOWPASS
- FMOD_DSP_HIGHPASS
- FMOD_DSP_ECHO
- FMOD_DSP_FLANGE
- FMOD_DSP_DISTORTION
- FMOD_DSP_NORMALIZE
- FMOD_DSP_LIMITER
- FMOD_DSP_PARAMEQ
- FMOD_DSP_PITCHSHIFT
- FMOD_DSP_CHORUS
- FMOD_DSP_ITECHO
- FMOD_DSP_COMPRESSOR
- FMOD_DSP_SFXREVERB
- FMOD_DSP_LOWPASS_SIMPLE
- FMOD_DSP_DELAY
- FMOD_DSP_TREMOLO
- FMOD_DSP_SEND
- FMOD_DSP_RETURN
- FMOD_DSP_HIGHPASS_SIMPLE
- FMOD_DSP_PAN
- FMOD_DSP_THREE_EQ
- FMOD_DSP_FFT
Firelight Technologies FMOD Studio API
DSP::getParameterInfo

Retrieve information about a specified parameter within the DSP unit.

C++ Syntax

FMOD_RESULT DSP::getParameterInfo(
    int index,
    FMOD_DSP_PARAMETER_DESC **desc
);

C Syntax

FMOD_RESULT FMOD_DSP_GetParameterInfo(
    FMOD_DSP *dsp,
    int index,
    FMOD_DSP_PARAMETER_DESC **desc
);

C# Syntax

RESULT DSP.getParameterInfo(
    int index,
    out DSP_PARAMETER_DESC desc
);

JavaScript Syntax

DSP.getParameterInfo(
    index,
    desc
); // writes value to desc.val
Parameters

index
Parameter index for this unit. Find the number of parameters with DSP::getNumParameters.

desc
Address of a variable to receive the contents of an array of FMOD_DSP_PARAMETER_DESC structures for 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 DSP::getNumParameters to find out the number of parameters for this DSP unit.
See Also

- DSP::setParameterFloat
- DSP::setParameterInt
- DSP::setParameterBool
- DSP::setParameterData
- DSP::getParameterFloat
- DSP::getParameterInt
- DSP::getParameterBool
- DSP::getParameterData
- DSP::getNumParameters
Firelight Technologies FMOD Studio API
DSP::getParameterInt

Retrieves a DSP unit's integer parameter by index. To find out the parameter names and range, see the see also field.

C++ Syntax

```cpp
FMOD_RESULT DSP::getParameterInt(
    int index,
    int *value,
    char *valuestr,
    int valuestrlen
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetParameterInt(
    FMOD_DSP *dsp,
    int index,
    int *value,
    char *valuestr,
    int valuestrlen
);
```

C# Syntax

```csharp
RESULT DSP.getParameterInt(
    int index,
    out int value
);
```

JavaScript Syntax

```javascript
DSP.getParameterInt(
    index,
    value, // writes value to value.val
    valuestr // writes value to valuestr.val
);
```
Parameters

index
Parameter index for this unit. Find the number of parameters with DSP::getNumParameters.

value
Address of a variable that receives the integer parameter value for the parameter specified.

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. Optional. Specify 0 or NULL to ignore.

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

The parameter properties can be retrieved with `DSP::getParameterInfo`.

JavaScript only:

Note: For the "valuestr" parameter, the maximum string length is 512.
See Also

- DSP::getParameterInfo
- DSP::getNumParameters
- DSP::setParameterInt
- FMOD_DSP_OSCILLATOR
- FMOD_DSP_LOWPASS
- FMOD_DSP_ITLOWPASS
- FMOD_DSP_HIGHPASS
- FMOD_DSP_ECHO
- FMOD_DSP_FLANGE
- FMOD_DSP_DISTORTION
- FMOD_DSP_NORMALIZE
- FMOD_DSP_LIMITER
- FMOD_DSP_PARAMEQ
- FMOD_DSP_PITCHSHIFT
- FMOD_DSP_CHORUS
- FMOD_DSP ITECHO
- FMOD_DSP_COMPRESSOR
- FMOD_DSP_SFXREVERB
- FMOD_DSP_LOWPASS_SIMPLE
- FMOD_DSP_DELAY
- FMOD_DSP_TREMOLLO
- FMOD_DSP_SEND
- FMOD_DSP_RETURN
- FMOD_DSP_HIGHPASS_SIMPLE
- FMOD_DSP_PAN
- FMOD_DSP THREE_EQ
- FMOD_DSP FFT
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT DSP.getSystemObject(
    out System system
);
```

**JavaScript Syntax**

```javascript
DSP.getSystemObject(  
    system             // writes value to system.val
);
```
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
- Channel::getDSP
- ChannelGroup::getDSP
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT DSP.getType(
    out DSP_TYPE type
);
```

**JavaScript Syntax**

```javascript
DSP.getType(
    type // writes value to type.val
);
```
Parameters

type
   Address of a variable to receive 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 \texttt{FMOD\_DSP\_TYPE\_UNKNOWN}.
See Also

- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT DSP.getUserData(
    out IntPtr userdata
);
```

JavaScript Syntax

```javascript
DSP.getUserData(
    userdata // writes value to userdata.val
);
```
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 Studio API
DSP::getWetDryMix

Retrieves the wet/dry scale of a DSP effect, through the 'wet' mix, which is the post-processed signal and the 'dry' mix which is the pre-processed signal.

C++ Syntax

```cpp
FMOD_RESULT DSP::getWetDryMix(
    float *prewet,
    float *postwet,
    float *dry
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_GetWetDryMix(
    FMOD_DSP *dsp,
    float *prewet,
    float *postwet,
    float *dry
);
```

C# Syntax

```csharp
RESULT DSP.getWetDryMix(
    out float prewet,
    out float postwet,
    out float dry
);
```

JavaScript Syntax

```javascript
DSP.getWetDryMix(  
    prewet,           // writes value to prewet.val
    postwet,          // writes value to postwet.val
    dry               // writes value to dry.val
);
```
**Parameters**

**prewet**
Address of a floating point value, to receive typically 0 to 1, describing a linear scale of the 'wet' (pre-processed signal) mix of the effect. Default = 1.0. Scale can be lower than 0 (negating) and higher than 1 (amplifying).

**postwet**
Address of a floating point value, to receive typically 0 to 1, describing a linear scale of the 'wet' (post-processed signal) mix of the effect. Default = 1.0. Scale can be lower than 0 (negating) and higher than 1 (amplifying).

**dry**
Address of a floating point value, to receive typically 0 to 1, describing a linear scale of the 'dry' (pre-processed signal) mix of the effect. Default = 0.0. Scale can be lower than 0 and higher than 1 (amplifying).
Return Values

If the function succeeds then the return value 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 dry signal path is silent by default, because dsp effects transform the input and pass the newly processed result to the output. It does not add to the input.
See Also

- DSP::setWetDryMix
Firelight Technologies FMOD Studio API
**DSP::release**

Frees a DSP object.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::release();
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_Release(FMOD_DSP *dsp);
```

**C# Syntax**

```csharp
RESULT DSP.release();
```

**JavaScript Syntax**

```javascript
DSP.release();
```
Return Values

If the function succeeds then the return value 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.

NOTE: If DSP is not removed from the Channel, ChannelGroup or System object with `Channel::removeDSP` or `ChannelGroup::removeDSP`, after being added with `Channel::addDSP` or `ChannelGroup::addDSP`, it will not release and will instead return `FMOD_ERR_DSP_INUSE`.
See Also

- System::createDSP
- System::createDSPByType
- Channel::getDSP
- ChannelGroup::getDSP
- Channel::addDSP
- ChannelGroup::addDSP
- Channel::removeDSP
- ChannelGroup::removeDSP
Firelight Technologies FMOD Studio API
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);
```

**C# Syntax**

```csharp
RESULT DSP.reset();
```

**JavaScript Syntax**

```javascript
DSP.reset();
```
Return Values

If the function succeeds then the return value 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 (ie 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 (ie clear and reset the echo filter) so that you dont get left over artifacts from the place it used to be connected.
Firelight Technologies FMOD Studio API
DSP::setActive

Enables or disables a unit for being processed.

C++ Syntax

FMOD_RESULT DSP::setActive(
    bool active
);

C Syntax

FMOD_RESULT FMOD_DSP_SetActive(
    FMOD_DSP *dsp,
    FMOD_BOOL active
);

C# Syntax

RESULT DSP.setActive(
    bool active
);

JavaScript Syntax

DSP.setActive(
    active
);

Parameters

active
  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.
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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT DSP.setBypass(
    bool bypass
);
```

**JavaScript Syntax**

```javascript
DSP.setBypass(
    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 Studio API
**DSP::setChannelFormat**

Sets the signal format of a dsp unit so that the signal is processed on the speakers specified. Also defines the number of channels in the unit that a read callback will process, and the output signal of the unit.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::setChannelFormat(
    FMOD_CHANNELMASK channelmask,
    int numchannels,
    FMOD_SPEAKERMODE source_speakermode
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_SetChannelFormat(
    FMOD_DSP *dsp,
    FMOD_CHANNELMASK channelmask,
    int numchannels,
    FMOD_SPEAKERMODE source_speakermode
);
```

**C# Syntax**

```csharp
RESULT DSP.setChannelFormat(
    CHANNELMASK channelmask,
    int numchannels,
    SPEAKERMODE source_speakermode
);
```

**JavaScript Syntax**

```javascript
DSP.setChannelFormat(
    channelmask,
    numchannels,
    source_speakermode
);
```
Parameters

channelmask
   A series of bits specified by FMOD_CHANNELMASK to determine which speakers are represented by the channels in the signal.

numchannels
   The number of channels to be processed on this unit and sent to the outputs connected to it. Maximum of FMOD_MAX_CHANNEL_WIDTH.

source_speakermode
   The source speaker mode where the signal came from. 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

Setting the number of channels on a unit will force a down or up mix to that channel count before processing the DSP read callback. This channel count is then sent to the outputs of the unit.

source_speakermode is informational, when channelmask describes what bits are active, and numchannels describes how many channels are in a buffer, source_speakermode describes where the channels originated from. For example if numchannels = 2 then this could describe for the DSP if the original signal started from a stereo signal or a 5.1 signal.

It could also describe the signal as all monaural, for example if numchannels was 16 and the speakermode was FMOD_SPEAKERMODE_MONO.
See Also

- FMOD_CHANNELMASK
- DSP::getChannelFormat
- FMOD_MAX_CHANNEL_WIDTH
Firelight Technologies FMOD Studio API
**DSP::setMeteringEnabled**

Enable metering for a DSP unit so that **DSP::getMeteringInfo** will return metering information, and so that FMOD Studio profiler tool can visualize the levels.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::setMeteringEnabled(
    bool inputEnabled,
    bool outputEnabled
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_SetMeteringEnabled(
    FMOD_DSP *dsp,
    FMOD_BOOL inputEnabled,
    FMOD_BOOL outputEnabled
);
```

**C# Syntax**

```csharp
RESULT DSP.setMeteringEnabled(
    bool inputEnabled,
    bool outputEnabled
);
```

**JavaScript Syntax**

```javascript
DSP.setMeteringEnabled(   inputEnabled,   outputEnabled  );
```
**Parameters**

inputEnabled
   Enable metering for the input signal (pre-processing). Specify true to turn on input level metering, false to turn it off.

outputEnabled
   Enable metering for the output signal (post-processing). Specify true to turn on output level metering, false to turn it off.
Return Values

If the function succeeds then the return value 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_INIT_PROFILE_METER_ALL with System::init will automatically turn on metering for all DSP units inside the FMOD mixer graph.
See Also

- DSP::getMeteringEnabled
- DSP::getMeteringInfo
- System::init
Firelight Technologies FMOD Studio API
DSP::setParameterBool

Sets a DSP unit's boolean parameter by index. To find out the parameter names and range, see the see also field.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::setParameterBool(
    int index,
    bool value
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_SetParameterBool(
    FMOD_DSP *dsp,
    int index,
    FMOD_BOOL value
);
```

**C# Syntax**

```csharp
RESULT DSPsetParameterBool(
    int index,
    bool value
);
```

**JavaScript Syntax**

```javascript
DSP.setParameterBool(
    index,
    value
);
```
Parameters

index
Parameter index for this unit. Find the number of parameters with DSP::getNumParameters.

value
Boolean parameter value to be passed to the DSP unit. Should be TRUE or 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

The parameter properties (such as min/max values) can be retrieved with DSP::getParameterInfo.
See Also

- DSP::getParameterInfo
- DSP::getNumParameters
- DSP::getParameterBool
- FMOD_DSP_OSCILLATOR
- FMOD_DSP_LOWPASS
- FMOD_DSP_ITLOWPASS
- FMOD_DSP_HIGHPASS
- FMOD_DSP_ECHO
- FMOD_DSP_FLANGE
- FMOD_DSP_DISTORTION
- FMOD_DSP_NORMALIZE
- FMOD_DSP_LIMITER
- FMOD_DSP_PARAMEQ
- FMOD_DSP_PITCHSHIFT
- FMOD_DSP_CHORUS
- FMOD_DSP_ITECHO
- FMOD_DSP_COMPRESSOR
- FMOD_DSP_SFXREVERB
- FMOD_DSP_LOWPASS_SIMPLE
- FMOD_DSP_DELAY
- FMOD_DSP_TREMOLO
- FMOD_DSP_SEND
- FMOD_DSP_RETURN
- FMOD_DSP_HIGHPASS_SIMPLE
- FMOD_DSP_PAN
- FMOD_DSP_THREE_EQ
- FMOD_DSP_FFT
Firelight Technologies FMOD Studio API
**DSP::setParameterData**

Sets a DSP unit's binary data parameter by index. To find out the parameter names and range, see the see also field.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::setParameterData(
    int index,
    void *data,
    unsigned int length
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_SetParameterData(
    FMOD_DSP *dsp,
    int index,
    void *data,
    unsigned int length
);
```

**C# Syntax**

```csharp
RESULT DSP.setParameterData(
    int index,
    byte[] data
);
```

**JavaScript Syntax**

```javascript
DSP.setParameterData(
    index,
    data,
    length
);
```
Parameters

index
Parameter index for this unit. Find the number of parameters with
\texttt{DSP::getNumParameters}.

data
Data block parameter. This will be raw binary data to be passed to the DSP
unit.

length
Length of data block in byte sbeing passed 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

The parameter properties (such as min/max values) can be retrieved with `DSP::getParameterInfo`.

Certain data types are predefined by the system and can be specified via the `FMOD_DSP_PARAMETER_DESC_DATA`, see `FMOD_DSP_PARAMETER_DATA_TYPE`
See Also

- DSP::getParameterInfo
- DSP::getNumParameters
- DSP::getParameterData
- FMOD_DSP_PARAMETER_DATA_TYPE
- FMOD_DSP_OSCILLATOR
- FMOD_DSP_LOWPASS
- FMOD_DSP_ITLOWPASS
- FMOD_DSP_HIGHPASS
- FMOD_DSP_ECHO
- FMOD_DSP_FLANGE
- FMOD_DSP_DISTORTION
- FMOD_DSP_NORMALIZE
- FMOD_DSP_LIMITER
- FMOD_DSP_PAREMEQ
- FMOD_DSP_PITCHSHIFT
- FMOD_DSP_CHORUS
- FMOD_DSP_ITECHO
- FMOD_DSP_COMPRESSOR
- FMOD_DSP_SFXREVERB
- FMOD_DSP_LOWPASS_SIMPLE
- FMOD_DSP_DELAY
- FMOD_DSP_TREMOLO
- FMOD_DSP_SEND
- FMOD_DSP_RETURN
- FMOD_DSP_HIGHPASS_SIMPLE
- FMOD_DSP_PAN
- FMOD_DSP_THREE_EQ
- FMOD_DSP_FFT
Firelight Technologies FMOD Studio API
DSP::setParameterFloat

Sets a DSP unit's floating point parameter by index. To find out the parameter names and range, see the see also field.

C++ Syntax

FMOD_RESULT DSP::setParameterFloat(
    int index,
    float value
);

C Syntax

FMOD_RESULT FMOD_DSP_SetParameterFloat(
    FMOD_DSP *dsp,
    int index,
    float value
);

C# Syntax

RESULT DSP.setParameterFloat(
    int index,
    float value
);

JavaScript Syntax

DSP.setParameterFloat(
    index,
    value
);
Parameters

index
   Parameter index for this unit. Find the number of parameters with
   \texttt{DSP::getNumParameters}.

value
   Floating point parameter value to be passed to the 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

The parameter properties (such as min/max values) can be retrieved with
DSP::getParameterInfo.
See Also

- DSP::getParameterInfo
- DSP::getNumParameters
- DSP::getParameterFloat
- FMOD_DSP_OSCILLATOR
- FMOD_DSP_LOWPASS
- FMOD_DSP_ITLOWPASS
- FMOD_DSP_HIGHPASS
- FMOD_DSP_ECHO
- FMOD_DSP_FLANGE
- FMOD_DSP_DISTORTION
- FMOD_DSP_NORMALIZE
- FMOD_DSP_LIMITER
- FMOD_DSP_PARAMEQ
- FMOD_DSP_PITCHSHIFT
- FMOD_DSP_CHORUS
- FMOD_DSP_ITECHO
- FMOD_DSP_COMPRESSOR
- FMOD_DSP_SFXREVERB
- FMOD_DSP_LOWPASS_SIMPLE
- FMOD_DSP_DELAY
- FMOD_DSP_TREMOLO
- FMOD_DSP_SEND
- FMOD_DSP_RETURN
- FMOD_DSP_HIGHPASS_SIMPLE
- FMOD_DSP_PAN
- FMOD_DSP_THREE_EQ
- FMOD_DSP_FFT
Firelight Technologies FMOD Studio API
DSP::setParameterInt

Sets a DSP unit's integer parameter by index. To find out the parameter names and range, see the see also field.

C++ Syntax

```cpp
FMOD_RESULT DSP::setParameterInt(
    int index,
    int value
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSP_SetParameterInt(
    FMOD_DSP *dsp,
    int index,
    int value
);
```

C# Syntax

```csharp
RESULT DSP.setParameterInt(
    int index,
    int value
);
```

JavaScript Syntax

```javascript
DSP.setParameterInt(
    index,
    value
);
```
Parameters

**index**
Parameter index for this unit. Find the number of parameters with `DSP::getNumParameters`.

**value**
Integer parameter value to be passed to the 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

The parameter properties (such as min/max values) can be retrieved with DSP::getParameterInfo.
See Also

- DSP::getParameterInfo
- DSP::getNumParameters
- DSP::getParameterInt
- FMOD_DSP_OSCILLATOR
- FMOD_DSP_LOWPASS
- FMOD_DSP_ITLOWPASS
- FMOD_DSP_HIGHPASS
- FMOD_DSP_ECHO
- FMOD_DSP_FLANGE
- FMOD_DSP_DISTORTION
- FMOD_DSP_NORMALIZE
- FMOD_DSP_LIMITER
- FMOD_DSP_PARAMEQ
- FMOD_DSP_PITCHSHIFT
- FMOD_DSP_CHORUS
- FMOD_DSP_ITECHO
- FMOD_DSP_COMPRESSOR
- FMOD_DSP_SFXREVERB
- FMOD_DSP_LOWPASS_SIMPLE
- FMOD_DSP_DELAY
- FMOD_DSP_TREMOLO
- FMOD_DSP_SEND
- FMOD_DSP_RETURN
- FMOD_DSP_HIGHPASS_SIMPLE
- FMOD_DSP_PAN
- FMOD_DSP_THREE_EQ
- FMOD_DSP_FFT
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT DSP.setUserData(
    IntPtr userdata
);
```

JavaScript Syntax

```javascript
DSP.setUserData(
    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 Studio API
**DSP::setWetDryMix**

Allows the user to scale the affect of a DSP effect, through control of the 'wet' mix, which is the post-processed signal and the 'dry' which is the pre-processed signal.

**C++ Syntax**

```cpp
FMOD_RESULT DSP::setWetDryMix(
    float prewet,
    float postwet,
    float dry
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_DSP_SetWetDryMix(
    FMOD_DSP *dsp,
    float prewet,
    float postwet,
    float dry
);
```

**C# Syntax**

```csharp
RESULT DSP.setWetDryMix(
    float prewet,
    float postwet,
    float dry
);
```

**JavaScript Syntax**

```javascript
DSP.setWetDryMix(
    prewet,
    postwet,
    dry
);
```
**Parameters**

**prewet**
Floating point value from 0 to 1, describing a linear scale of the 'wet' (pre-processed signal) mix of the effect. Default = 1.0. Scale can be lower than 0 (negating) and higher than 1 (amplifying).

**postwet**
Floating point value from 0 to 1, describing a linear scale of the 'wet' (post-processed signal) mix of the effect. Default = 1.0. Scale can be lower than 0 (negating) and higher than 1 (amplifying).

**dry**
Floating point value from 0 to 1, describing a linear scale of the 'dry' (pre-processed signal) mix of the effect. Default = 0.0. Scale can be lower than 0 and higher than 1 (amplifying).
Return Values

If the function succeeds then the return value 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 dry signal path is silent by default, because dsp effects transform the input and pass the newly processed result to the output. It does not add to the input.
See Also

- **DSP::getWetDryMix**
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT DSP.showConfigDialog(
    IntPtr hwnd,
    bool show
);
```

**JavaScript Syntax**

```javascript
DSP.showConfigDialog(
    hwnd,
    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 \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

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::setParameterFloat` / `DSP::setParameterInt` / `DSP::setParameterBool` / `DSP::setParameterData`.

These are usually VST plugins. FMOD Studio 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::setParameterFloat
- DSP::setParameterInt
- DSP::setParameterBool
- DSP::setParameterData
- DSP::getParameterFloat
- DSP::getParameterInt
- DSP::getParameterBool
- DSP::getParameterData
Firelight Technologies FMOD Studio API
DSPConnection
Functions

DSPConnection::getInput  DSPConnection::getMix
DSPConnection::getMixMatrix
DSPConnection::getOutput
DSPConnection::getType
DSPConnection::getUserData
DSPConnection::setMix
DSPConnection::setMixMatrix
DSPConnection::setUserData
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT DSPConnection.getInput(
    out DSP input
);
```

**JavaScript Syntax**

```javascript
DSPConnection.getInput(
    input // writes value to input.val
);
```
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 Studio API
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
);
```

C# Syntax

```csharp
RESULT DSPConnection.getMix(
    out float volume
);
```

JavaScript Syntax

```javascript
DSPConnection.getMix(
    volume // writes value to volume.val
);
```
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 \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

- DSPConnection::setMix
- DSP::getInput
- DSP::getOutput
Firelight Technologies FMOD Studio API
DSPConnection::getMixMatrix

Returns the panning matrix set by the user, for a connection.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::getMixMatrix(
    float *matrix,
    int *outchannels,
    int *inchannels,
    int inchannel_hop
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_GetMixMatrix(
    FMOD_DSPCONNECTION *dspconnection,
    float *matrix,
    int *outchannels,
    int *inchannels,
    int inchannel_hop
);
```

C# Syntax

```csharp
RESULT DSPConnection.getMixMatrix(
    float[] matrix, // writes value to matrix.val
    out int outchannels, // writes value to outchannels.val
    out int inchannels, // writes value to inchannels.val
    int inchannel_hop
);
```

JavaScript Syntax

```javascript
DSPConnection.getMixMatrix(
    matrix, // writes value to matrix.val
    outchannels, // writes value to outchannels.val
    inchannels, // writes value to inchannels.val
    inchannel_hop
);
```
Parameters

matrix
   Address of a variable to receive an array of floating point matrix data, where rows represent output speakers, and columns represent input channels.

outchannels
   Address of a variable to receive the number of output channels in the set matrix.

inchannels
   Address of a variable to receive the number of input channels in the set matrix.

inchannel_hop
   Number of floating point values available in the destination memory for a row, so that the destination memory can be skipped through correctly to write the right values, if the intended matrix memory to be written to is wider than the matrix stored in the DSPConnection.
Return Values

If the function succeeds then the return 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::setMixMatrix`
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULT DSPConnection.getOutput(  
    out DSP output
);
```

JavaScript Syntax

```javascript
DSPConnection.getOutput(  
    output                                             // writes value to output.val
);
```
**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 Studio API
DSPConnection::getType

Returns the type of the connection between 2 DSP units. This can be
FMOD_DSPCONNECTION_TYPE_STANDARD,
FMOD_DSPCONNECTION_TYPE_SIDECHAIN,
FMOD_DSPCONNECTION_TYPE_SEND or
FMOD_DSPCONNECTION_TYPE_SEND_SIDECHAIN.

C++ Syntax

FMOD_RESULT DSPConnection::getType(
    FMOD_DSPCONNECTION_TYPE *type
);

C Syntax

FMOD_RESULT FMOD_DSPConnection_GetType(
    FMOD_DSPCONNECTION *dspconnection,
    FMOD_DSPCONNECTION_TYPE *type
);

C# Syntax

RESULT DSPConnection.getType(
    out DSPCONNECTION_TYPE type
);

JavaScript Syntax

DSPConnection.getType(
    type // writes value to type.val
);
Parameters

type
    Address of the variable to receive the type of connection between 2 DSP units. See FMOD_DSPCONNECTION_TYPE.
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_DSPCONNECTION_TYPE`
Firelight Technologies FMOD Studio API
**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
);
```

**C# Syntax**

```csharp
RESULT DSPConnection.getUserData(
    out IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
DSPConnection.getUserData(
    userdata // writes value to userdata.val
);
```
**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 Studio API
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
);
```

C# Syntax

```csharp
RESULT DSPConnection.setMix(
    float volume
);
```

JavaScript Syntax

```javascript
DSPConnection.setMix(
    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 Studio API
DSPConnection::setMixMatrix

Sets a NxN panning matrix on a DSP connection. Skipping/hop is supported, so memory for the matrix can be wider than the width of the inchannels parameter.

C++ Syntax

```cpp
FMOD_RESULT DSPConnection::setMixMatrix(
    float *matrix,
    int outchannels,
    int inchannels,
    int inchannel_hop
);
```

C Syntax

```c
FMOD_RESULT FMOD_DSPConnection_SetMixMatrix(
    FMOD_DSPCONNECTION *dspconnection,
    float *matrix,
    int outchannels,
    int inchannels,
    int inchannel_hop
);
```

C# Syntax

```csharp
RESULT DSPConnection.setMixMatrix(
    float[] matrix,
    int outchannels,
    int inchannels,
    int inchannel_hop
);
```

JavaScript Syntax

```javascript
DSPConnection.setMixMatrix(
    matrix,
    outchannels,
    inchannels,
    inchannel_hop
);
```
Parameters

matrix
   Pointer to an array of floating point matrix data, where rows represent output speakers, and columns represent input channels.
outchannels
   Number of output channels in the matrix being specified.
inchannels
   Number of input channels in the matrix being specified.
inchannel_hop
   Number of floating point values stored in memory for a row, so that the memory can be skipped through correctly to read the right values, if the intended matrix memory to be read from is wider than the matrix stored in the DSPConnection.
Return Values

If the function succeeds then the return 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::getMixMatrix
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT DSPConnection.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
DSPConnection.setUserData(
    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 Studio API
Geometry
Functions

Geometry::addPolygon  Geometry::getActive
Geometry::getMaxPolygons
Geometry::getNumPolygons
Geometry::getPolygonAttributes
Geometry::getPolygonNumVertices
Geometry::getPolygonVertex
Geometry::getPosition
Geometry::getRotation
Geometry::setScale
Geometry::getUserData
Geometry::release
Geometry::save
Geometry::setActive
Geometry::setPolygonAttributes
Geometry::setPolygonVertex
Geometry::setPosition
Geometry::setRotation
Geometry::setScale
Geometry::setUserData
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
RESULTGeometry.addPolygon(
    float directocclusion,
    float reverbocclusion,
    bool doublesided,
    int numvertices,
    VECTOR[] vertices,
    out int polygonindex
);
```

JavaScript Syntax

```javascript
Geometry.addPolygon();
```
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 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.

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.

JavaScript only:

This function is not currently supported
See Also

- Geometry::getNumPolygons
- Geometry::setPosition
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Geometry.getActive(
    out bool active
);
```

**JavaScript Syntax**

```javascript
Geometry.getActive(
    active // writes value to active.val
);
```
**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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Geometry.getMaxPolygons(
    out int maxpolygons,
    out int maxvertices
);
```

JavaScript Syntax

```javascript
Geometry.getMaxPolygons(
    maxpolygons, // writes value to maxpolygons.val
    maxvertices,  // writes value to maxvertices.val
);
```
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 Studio API
Geometry::getNumPolygons

Retrieves the number of polygons stored within this geometry object.

C++ Syntax

FMOD_RESULT Geometry::getNumPolygons(
    int *numpolygons
);

C Syntax

FMOD_RESULT FMOD_Geometry_GetNumPolygons(
    FMOD_GEOMETRY *geometry,
    int *numpolygons
);

C# Syntax

RESULT Geometry.getNumPolygons(
    out int numpolygons
);

JavaScript Syntax

Geometry.getNumPolygons(
    numpolygons // writes value to 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
Firelight Technologies FMOD Studio API
Geometry::getPolygonAttributes

Retrieves the attributes for a particular polygon inside a geometry object.

C++ Syntax

FMOD_RESULT Geometry::getPolygonAttributes(
    int index,
    float *directocclusion,
    float *reverbocclusion,
    bool *doublesided
);

C Syntax

FMOD_RESULT FMOD_Geometry_GetPolygonAttributes(
    FMOD_GEOMETRY *geometry,
    int index,
    float *directocclusion,
    float *reverbocclusion,
    FMOD_BOOL *doublesided
);

C# Syntax

RESULT Geometry.getPolygonAttributes(
    int index,
    out float directocclusion,
    out float reverbocclusion,
    out bool doublesided
);

JavaScript Syntax

Geometry.getPolygonAttributes(
    index,
    directocclusion,       // writes value to directocclusion
    reverbocclusion,       // writes value to reverbocclusion
    doublesided           // writes value to doublesided.value
);
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 polygon fully occludes reverb (reverb reflections will be silent through this polygon).
doublesided
   Address of a variable to receive 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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Geometry.getPolygonNumVertices(
   int index,
   out int numvertices
);
```

**JavaScript Syntax**

```javascript
Geometry.getPolygonNumVertices(
   index,
   numvertices // writes value to 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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Geometry.getPolygonVertex(
    int index,
    int vertexindex,
    out VECTOR vertex
);
```

**JavaScript Syntax**

```javascript
Geometry.getPolygonVertex(
    index,
    vertexindex,
    vertex // writes value to vertex.val
);
```
Parameters

index
Polygons index. This must be in the range of 0 to $\text{Geometry::getNumPolygons}$ minus 1.

vertexindex
Vertex index inside the polygon. This must be in the range of 0 to $\text{Geometry::getPolygonNumVertices}$ 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
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Geometry.getPosition(
    out VECTOR position
);
```

**JavaScript Syntax**

```javascript
Geometry.getPosition( 
    position // writes value to position.val 
);
```
**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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Geometry.getRotation(
    out VECTOR forward,
    out VECTOR up
);
```

**JavaScript Syntax**

```javascript
Geometry.getRotation(
    forward, \n    up \n); // writes value to forward.val
    up \n); // writes value to up.val
```
**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::set3DListenerAttributes
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
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**

```cpp
FMOD_RESULT Geometry::getScale(
    FMOD_VECTOR *scale
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Geometry_GetScale(
    FMOD_GEOMETRY *geometry,
    FMOD_VECTOR *scale
);
```

**C# Syntax**

```csharp
RESULT Geometry.getScale(
    out VECTOR scale
);
```

**JavaScript Syntax**

```javascript
Geometry.getScale(
    scale // writes value to scale.val
);
```
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 \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

- Geometry::setScale
- FMOD_VECTOR
Firelight Technologies FMOD Studio API
Geometry::getUserData

Retrieves the user value that 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
);
```

**C# Syntax**

```csharp
RESULT Geometry.getUserData(
    out IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
Geometry.getUserData(
    userdata // writes value to userdata.val
);
```
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 Studio API
Geometry::release

Frees a geometry object and releases its memory.

C++ Syntax

FMOD_RESULT Geometry::release();

C Syntax

FMOD_RESULT FMOD_Geometry_Release(FMOD_GEOMETRY *geometry);

C# Syntax

RESULT Geometry.release();

JavaScript Syntax

Geometry.release();
Return Values

If the function succeeds then the return 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 Studio API
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

FMOD_RESULT Geometry::save(
    void *data,
    int *datasize
);

C Syntax

FMOD_RESULT FMOD_Geometry_Save(
    FMOD_GEOMETRY *geometry,
    void *data,
    int *datasize
);

C# Syntax

RESULT Geometry.save(
    IntPtr data,
    out int datasize
);

JavaScript Syntax

Geometry.save(
);
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 **FMOD_RESULT** enumeration.
**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.

**JavaScript only:**

This function is not currently supported
See Also

- `System::loadGeometry`
- `System::createGeometry`
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Geometry.SetActive(
    bool active
);
```

**JavaScript Syntax**

```javascript
Geometry.setActive(
    active
);
```
Parameters

active
   true = active, false = not active. Default = true.
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

- `Geometry::getActive`
Firelight Technologies FMOD Studio API
Geometry::setPolygonAttributes

Sets individual attributes for each polygon inside a geometry object.

C++ Syntax

```
FMOD_RESULT Geometry::setPolygonAttributes(
    int index,
    float directocclusion,
    float reverbocclusion,
    bool doublesided
);
```

C Syntax

```
FMOD_RESULT FMOD_Geometry_SetPolygonAttributes(
    FMOD_GEOMETRY *geometry,
    int index,
    float directocclusion,
    float reverbocclusion,
    FMOD_BOOL doublesided
);
```

C# Syntax

```
RESULT Geometry.setPolygonAttributes(
    int index,
    float directocclusion,
    float reverbocclusion,
    bool doublesided
);
```

JavaScript Syntax

```
Geometry.setPolygonAttributes(
    index,
    directocclusion,
    reverbocclusion,
    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 Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Geometry.setPolygonVertex(
    int index,
    int vertexindex,
    ref VECTOR vertex
);
```

**JavaScript Syntax**

```javascript
Geometry.setPolygonVertex(
    index,
    vertexindex,
    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 Studio API
Geometry::setPosition

Sets the position of the object in world space, which is the same space FMOD sounds and listeners reside in.

C++ Syntax

FMOD_RESULT Geometry::setPosition(
    const FMOD_VECTOR *position
);

C Syntax

FMOD_RESULT FMOD_Geometry_SetPosition(
    FMOD_GEOMETRY *geometry,
    const FMOD_VECTOR *position
);

C# Syntax

RESULT Geometry.setPosition(
    ref VECTOR position
);

JavaScript Syntax

Geometry.setPosition(
    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 Studio API
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
);
```

C# Syntax

```csharp
RESULT Geometry.setRotation(
    ref VECTOR forward,
    ref VECTOR up
);
```

JavaScript Syntax

```javascript
Geometry.setRotation(
    forward,
    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 \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 remarks in System::set3DListenerAttributes for more description on forward and up vectors.
See Also

- Geometry::getRotation
- System::set3DListenerAttributes
- FMOD VECTOR
Firelight Technologies FMOD Studio API
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
);
```

**C# Syntax**

```csharp
RESULT Geometry.setScale(
    ref VECTOR scale
);
```

**JavaScript Syntax**

```javascript
Geometry.setScale(
    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 Studio API
Geometry::setUserData

Sets a user value that the Geometry object will store internally. Can be retrieved with Geometry::getUserData.

C++ Syntax

FMOD_RESULT Geometry::setUserData(
    void *userdata
);

C Syntax

FMOD_RESULT FMOD_Geometry_SetUserData(
    FMOD_GEOMETRY *geometry,
    void *userdata
);

C# Syntax

RESULT Geometry.setUserData(
    IntPtr userdata
);

JavaScript Syntax

Geometry.setUserData(
    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 Studio API
Reverb3D
Functions

Reverb3D::get3DAttributes  Reverb3D::getActive
Reverb3D::getProperties
Reverb3D::getUserData
Reverb3D::release
Reverb3D::set3DAttributes
Reverb3D::setActive
Reverb3D::setProperties
Reverb3D::setUserData
Firelight Technologies FMOD Studio API
Reverb3D::get3DAttributes

Retrieves the 3D attributes of a Reverb object.

C++ Syntax

```cpp
FMOD_RESULT Reverb3D::get3DAttributes(
    FMOD_VECTOR *position,
    float *mindistance,
    float *maxdistance
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb3D_Get3DAttributes(
    FMOD_REVERB3D *reverb3d,
    FMOD_VECTOR *position,
    float *mindistance,
    float *maxdistance
);
```

C# Syntax

```csharp
RESULT Reverb3D.get3DAttributes(
    ref VECTOR position,
    ref float mindistance,
    ref float maxdistance
);
```

JavaScript Syntax

```javascript
Reverb3D.get3DAttributes(
    position, // writes value to position.val
    mindistance, // writes value to mindistance.val
    maxdistance // writes value to maxdistance.val
);
```
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

- `Reverb3D::set3DAttributes`
- `System::createReverb3D`
Firelight Technologies FMOD Studio API
Reverb3D::getActive

Retrieves the active state of the reverb object.

**C++ Syntax**

```cpp
FMOD_RESULT Reverb3D::getActive(
    bool *active
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Reverb3D_GetActive(
    FMOD_REVERB3D *reverb3d,
    FMOD_BOOL *active
);
```

**C# Syntax**

```csharp
RESULT Reverb3D.getActive(
    out bool active
);
```

**JavaScript Syntax**

```javascript
Reverb3D.getActive(
    active
); // writes value to active.val
```
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

- `Reverb3D::setActive`
- `System::createReverb3D`
Firelight Technologies FMOD Studio API
Reverb3D::getProperties

Retrieves the current reverb environment.

**C++ Syntax**

```cpp
FMOD_RESULT Reverb3D::getProperties(
    FMOD_REVERB_PROPERTIES *properties
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Reverb3D_GetProperties(
    FMOD_REVERB3D *reverb3d,
    FMOD_REVERB_PROPERTIES *properties
);
```

**C# Syntax**

```csharp
RESULT Reverb3D.getProperties(
    ref REVERB_PROPERTIES properties
);
```

**JavaScript Syntax**

```javascript
Reverb3D.getProperties(
    properties // writes value to properties.val
);
```
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

- Reverb3D::setProperties
- System::createReverb3D
Firelight Technologies FMOD Studio API
Reverb3D::getUserData

Retrieves the user value that that was set by calling the Reverb::setUserData function.

C++ Syntax

```cpp
FMOD_RESULT Reverb3D::getUserData(
    void **userdata
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb3D_GetUserData(
    FMOD_REVERB3D *reverb3d,
    void **userdata
);
```

C# Syntax

```csharp
RESULT Reverb3D.getUserData(
    out IntPtr userdata
);
```

JavaScript Syntax

```javascript
Reverb3D.getUserData(
    userdata // writes value to userdata.val
);
```
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 \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

- Reverb3D::setUserData
Firelight Technologies FMOD Studio API
Reverb3D::release

Releases the memory for a reverb object and makes it inactive.

C++ Syntax

FMOD_RESULT Reverb3D::release();

C Syntax

FMOD_RESULT FMOD_Reverb3D_Release(FMOD_REVERB3D *reverb3d);

C# Syntax

RESULT Reverb3D.release();

JavaScript Syntax

Reverb3D.release();
Return Values

If the function succeeds then the return value 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::createReverb3D
Firelight Technologies FMOD Studio API
Reverb3D::set3DAttributes

Sets the 3D properties of a 'virtual' reverb object.

**C++ Syntax**

```cpp
FMOD_RESULT Reverb3D::set3DAttributes(
    const FMOD_VECTOR *position,
    float mindistance,
    float maxdistance
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Reverb3D_Set3DAttributes(
    FMOD_REVERB3D *reverb3d,
    const FMOD_VECTOR *position,
    float mindistance,
    float maxdistance
);
```

**C# Syntax**

```csharp
RESULT Reverb3D.set3DAttributes(
    ref VECTOR position,
    float mindistance,
    float maxdistance
);
```

**JavaScript Syntax**

```javascript
Reverb3D.set3DAttributes(
    position,
    mindistance,
    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.

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

- Reverb3D::get3DAttributes
- System::createReverb3D
Firelight Technologies FMOD Studio API
Reverb3D::setActive

Disables or enables a reverb object so that it does or does not contribute to the 3D scene.

C++ Syntax

```cpp
FMOD_RESULT Reverb3D::setActive(
    bool active
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb3D_SetActive(
    FMOD_REVERB3D *reverb3d,
    FMOD_BOOL active
);
```

C# Syntax

```csharp
RESULT Reverb3D.SetActive(
    bool active
);
```

JavaScript Syntax

```javascript
Reverb3D.setActive(
    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

- Reverb3D::setActive
- System::createReverb3D
Firelight Technologies FMOD Studio API
Reverb3D::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_common.h header.

C++ Syntax

```cpp
FMOD_RESULT Reverb3D::setProperties(
    const FMOD_REVERB_PROPERTIES *properties
);
```

C Syntax

```c
FMOD_RESULT FMOD_Reverb3D_SetProperties(
    FMOD_REVERB3D *reverb3d,
    const FMOD_REVERB_PROPERTIES *properties
);
```

C# Syntax

```csharp
RESULT Reverb3D.setProperties(
    ref REVERB_PROPERTIES properties
);
```

JavaScript Syntax

```javascript
Reverb3D.setProperties(
    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
- FMOD_REVERB_PRESETS
- Reverb3D::getProperties
- System::createReverb3D
Firelight Technologies FMOD Studio API
**Reverb3D::setUserData**

Sets a user value that the Reverb object will store internally. Can be retrieved with Reverb::getUserData.

**C++ Syntax**

```cpp
FMOD_RESULT Reverb3D::setUserData(
    void *userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Reverb3D_SetUserData(
    FMOD_REVERB3D *reverb3d,
    void *userdata
);
```

**C# Syntax**

```csharp
RESULT Reverb3D.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
Reverb3D.setUserData(
    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

- Reverb3D::getUserData
Firelight Technologies FMOD Studio API
Functions

>ErrorString file_close
d file_open
d file_read
d file_seek
Debug_Initialize
File_GetDiskBusy
File_SetDiskBusy
Memory_GetStats
Memory_Initialize
System_Create
FS_createPreloadedFile
Memory_Free
ReadFile
setValue
Firelight Technologies FMOD Studio API
**ErrorString**

Returns a more verbose string version of the error code returned by any FMOD function.

**JavaScript Syntax**

```javascript
FMOD.ErrorString(
    error // FMOD_RESULT error code
);
```
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.
Firelight Technologies FMOD Studio API
file_close

Helper function to close a file manually, that is preloaded with FMOD.FS_createPreloadedFile

JavaScript Syntax

```javascript
FMOD.file_close(
    handle,                 // (object) Handle returned by the FMOD.file_o
);
```
Return Values

If the function succeeds then the return 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 Studio API
file_open

Helper function to open a file manually, that is preloaded with FMOD.FS_createPreloadedFile

JavaScript Syntax

```javascript
FMOD.file_open(
    system, // FMOD::System object handle
    filename, // (string) path and filename which matches th
    filesize_out, // (number) an integer with the size of the fi
    handle_out // (object) an object with the file handle is
);
```
Return Values

If the function succeeds then the return 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 Studio API
file_read

Helper function to read a file manually, that is preloaded with FMOD.FS_createPreloadedFile

JavaScript Syntax

```javascript
FMOD.file_read(
    handle, // (object) Handle returned by the FMOD.file_open
    buffer, // (number) A memory address that would come from
    sizebytes, // (number) Integer value with the number of bytes
    bytesRead_out // (number) An integer with the number of bytes ac
);
```
Return Values

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

'buffer' parameter is typically an object returned as the
1. buffer parameter of the FMOD_FILE_READ_CALLBACK
2. data parameter of the FMOD_CREATESOUNDEXINFO::pcmreadcallback,
3. buffer parameter of the FMOD_STUDIO_BANK_INFO::readcallback
Firelight Technologies FMOD Studio API
file_seek

Helper function to seek a file manually, that is preloaded with FMOD.FS_createPreloadedFile

JavaScript Syntax

```javascript
FMOD.file_seek(
    handle,    // (object) Handle returned by the FMOD.file_open func
    pos,       // (number) offset in bytes to seek into the file, rel
);
```
Return Values

If the function succeeds then the return 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 Studio API
Debug_Initialize

Specify the level and delivery method of log messages when using the logging version of FMOD.

**C++ Syntax**

```cpp
FMOD_RESULT Debug_Initialize(
    FMOD_DEBUG_FLAGS flags,
    FMOD_DEBUG_MODE mode,
    FMOD_DEBUG_CALLBACK callback,
    const char *filename
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Debug_Initialize(
    FMOD_DEBUG_FLAGS flags,
    FMOD_DEBUG_MODE mode,
    FMOD_DEBUG_CALLBACK callback,
    const char *filename
);
```

**C# Syntax**

```csharp
static RESULT Debug.Initialize(
    DEBUG_FLAGS flags,
    DEBUG_MODE mode,
    DEBUG_CALLBACK callback,
    string filename
);
```

**JavaScript Syntax**

```javascript
FMOD.Debug.Initialize(
    flags,
);
```
Parameters

flags

Mask of bits representing the desired log information. Note: LOG implies WARN and WARN implies ERROR.

mode

Destination for log messages.

callback

Callback to use when mode is set to callback, only required when using that mode.

filename

Filename to use when mode is set to file, only required when using that 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.
Remarks

This function will return **FMOD_ERR_UNSUPPORTED** when using the non-logging (release) versions of FMOD.

The logging version of FMOD can be recognized by the 'L' suffix in the library name, fmodL.dll or libfmodL.so for instance.
See Also

- FMOD_DEBUG_FLAGS
- FMOD_DEBUG_MODE
- FMOD_DEBUG_CALLBACK
Firelight Technologies FMOD Studio API
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 Studio API
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 simulatenous 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 Studio API
Memory_GetStats

Returns information on the memory usage of FMOD. This information is byte accurate and counts all allocs and frees internally.

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
);
```

C# Syntax

```csharp
static RESULT Memory.GetStats(
    out int currentallocated,
    out int maxallocated,
    bool blocking
);
```

JavaScript Syntax

```javascript
FMOD.Memory_GetStats(
    currentallocated,,
    maxallocated,,
    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

This is useful for determining a fixed memory size to make FMOD work within for fixed memory machines such as consoles.

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_Init`
Firelight Technologies FMOD Studio API
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_ALLOC_CALLBACK useralloc,
   FMOD_MEMORY_REALLOC_CALLBACK userrealloc,
   FMOD_MEMORY_FREE_CALLBACK userfree,
   FMOD_MEMORY_TYPE memtypeflags
);
```

C Syntax

```c
FMOD_RESULT FMOD_Memory_Initialize(
   void *poolmem,
   int poollen,
   FMOD_MEMORY_ALLOC_CALLBACK useralloc,
   FMOD_MEMORY_REALLOC_CALLBACK userrealloc,
   FMOD_MEMORY_FREE_CALLBACK userfree,
   FMOD_MEMORY_TYPE memtypeflags
);
```

C# Syntax

```csharp
static RESULT Memory.Initialize(
   IntPtr poolmem,
   int poollen,
   MEMORY_ALLOC_CALLBACK useralloc,
   MEMORY_REALLOC_CALLBACK userrealloc,
   MEMORY_FREE_CALLBACK userfree,
   MEMORY_TYPE memtypeflags
);
```

JavaScript Syntax
FMOD.Memory_Initialize(
    poolmem,,
    poollen,,
    useralloc,,
    userrealloc,,
    userfree,,
    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 poolmem. 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. Bitwise OR these together for multiple 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

This function must be called before any FMOD System object is created.

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 mainly does allocation when creating streams, music or samples and the System::init stage. It will rarely allocate or deallocate 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, FMOD_MEMORY_ALL)
FMOD::Memory_Initialize(NULL, 0, myalloc, myrealloc, myfree, FMOD_MEMORY_ALL)
FMOD::Memory_Initialize(ptr, len, NULL, NULL, NULL, FMOD_MEMORY_ALL)

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 thread safe 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_ALLOC_CALLBACK`
- `FMOD_MEMORY_REALLOC_CALLBACK`
- `FMOD_MEMORY_FREE_CALLBACK`
- `Memory_GetStats`
Firelight Technologies FMOD Studio API
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
);
```

C# Syntax

```csharp
static RESULT Factory.System_Create(
    out System system
);
```

JavaScript Syntax

```javascript
FMOD.System_Create(
    system,
);
```
Parameters

\textit{system}

Address of a pointer that receives the new FMOD System object. \textbf{HTML5 Note} - the object is written to system.val
Return Values

If the function succeeds then the return value 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.

NOTE! Calls to System_Create and System::release are not thread-safe. Do not call these functions simultaneously from multiple threads at once.
See Also

- System::init
- System::release
Firelight Technologies FMOD Studio API
FS_createPreloadedFile

Mounts a local file so that FMOD can recognize it when calling a function that uses a filename (ie loadBank/createSound)
See https://kripken.github.io/emscripten-site/docs/api_reference/FileSystem-API.html#FS.createPreloadedFile for docs on FS_createPreloadedFile

JavaScript Syntax

```javascript
FMOD.FS_createPreloadedFile(
    foldername,    // (string). Parent folder, ie '/'
    filename,      // (string). Filename to preload.
    url,           // (string). Path inside parent folder. ie canread,
    canwrite       // (boolean). Whether the file should have canread,
);                  // (boolean). Whether the file should have
```
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.
Firelight Technologies FMOD Studio API
Memory_Free

Frees memory that was allocated by FMOD internally.

JavaScript Syntax

```javascript
FMOD.Memory_Free(
    memory,             // Memory object supplied by a previous FM
);
```
Return Values

If the function succeeds then the return value 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!** Currently **FMOD.ReadFile** is the only function that returns an object with memory allocated by FMOD.
Firelight Technologies FMOD Studio API
ReadFile

Read the entire contents of a file into memory, as a JS variable that contains nothing but a memory address.

JavaScript Syntax

```javascript
FMOD.ReadFile(
    system,   // FMOD::System object handle
    filename, // (string) Filename of the file that is to be loaded
    output,   // The variable with the allocated memory containing the file contents
);
```
Return Values

If the function succeeds then the return value 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 filename can be used to load a file that has been preloaded with FMOD.FS_createPreloadedFile

Note! The memory for the file is allocated internally in FMOD from the JS heap, so needs to be freed with FMOD.Memory_Free when not needed any more.
Firelight Technologies FMOD Studio API
setValue

Writes a value to memory managed by FMOD. This sort of value is typically sound data that is passed to the user as an 'address' which is internal to FMOD. This avoids duplication of large buffers which waste memory, because everything is passed by reference in JS.

**JavaScript Syntax**

```javascript
FMOD.setValue(
    address,  // Memory address returned by an FMOD function.
    value,   // (number) A value which can be an integer or a real/floating point number.
    format   // (string) A format 'string' which identifies
);  
```
Return Values

If the function succeeds then the return value 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 https://kripken.github.io/emscripten-site/docs/api_reference/preamble.js.html#accessing-memory for docs on setValue.

'address' is a memory address, and only FMOD functions will return a memory address. Examples of this would be

1. buffer parameter of the FMOD_FILE_READ_CALLBACK
2. data parameter of the FMOD_CREATESOUNDEXINFO::pcmreadcallback,
3. buffer parameter of the FMOD_STUDIO_BANK_INFO::readcallback

'format' can be values like 'i8', 'i16', 'i32', 'i64', 'float', 'double' typically.
Firelight Technologies FMOD Studio API
Callbacks

FMOD_3D_ROLLOFF_CALLBACK
FMOD_CHANNELCONTROL_CALLBACK
FMOD_CODEC_CLOSE_CALLBACK
FMOD_CODEC_GETLENGTH_CALLBACK
FMOD_CODEC_GETPOSITION_CALLBACK
FMOD_CODEC_METADATA_CALLBACK
FMOD_CODEC_OPEN_CALLBACK
FMOD_CODEC_READ_CALLBACK
FMOD_CODEC_SETPOSITION_CALLBACK
FMOD_CODEC_SOUNDCREATE_CALLBACK
FMOD_DEBUG_CALLBACK
FMOD_DSP_CREATE_CALLBACK
FMOD_DSP_DIALOG_CALLBACK
FMOD_DSP_GETPARAM_BOOL_CALLBACK
FMOD_DSP_GETPARAM_DATA_CALLBACK
FMOD_DSP_GETPARAM_FLOAT_CALLBACK
FMOD_DSP_GETPARAM_INT_CALLBACK
FMOD_DSP_PROCESS_CALLBACK
FMOD_DSP_READ_CALLBACK
FMOD_DSP_RELEASE_CALLBACK
FMOD_DSP_RESET_CALLBACK
FMOD_DSP_SETPARAM_BOOL_CALLBACK
FMOD_DSP_SETPARAM_DATA_CALLBACK
FMOD_DSP_SETPARAM_FLOAT_CALLBACK
FMOD_DSP_SETPARAM_INT_CALLBACK
FMOD_DSP_SETPOSITION_CALLBACK
FMOD_DSP_SHOULDIPROCESS_CALLBACK
FMOD_DSP_SYSTEM_DEREGISTER_CALLBACK
FMOD_DSP_SYSTEM_MIX_CALLBACK
FMOD_DSP_SYSTEM_REGISTER_CALLBACK
FMOD_FILE_ASYNCCANCEL_CALLBACK
FMOD_FILE_ASYNCDONE_FUNC
FMOD_FILE_ASYNCREAD_CALLBACK
FMOD_FILE_CLOSE_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_3D_ROLLOFF_CALLBACK

Callback for system wide 3D channel volume calculation which overrides fmod's internal calculation code.

C/C++ Syntax

```c
float F_CALLBACK FMOD_3D_ROLLOFF_CALLBACK(
    FMOD_CHANNEL *channel,
    float distance
);
```
Parameters

channel

Pointer to a channel handle.

distance

Distance in units (meters by default).
Return Values

Return the channel volume for the given distance.
Remarks

C++ Users. Cast FMOD_CHANNEL * to FMOD::Channel * inside the callback and use as normal.
See Also

- System::set3DRolloffCallback
- System::set3DListenerAttributes
- System::get3DListenerAttributes
- Channel::getUserData
Firelight Technologies FMOD Studio API
FMOD_CHANNELCONTROL_CALLBACK

Callback for channel events.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_CHANNELCONTROL_CALLBACK(
    FMOD_CHANNELCONTROL *channel,
    FMOD_CHANNELCONTROL_TYPE controltype,
    FMOD_CHANNELCONTROL_CALLBACK_TYPE callbacktype,
    void *commanddata1,
    void *commanddata2
);
```
**Parameters**

*channel*

Pointer to a channel control handle.

*controltype*

Subtype of the channel control handle, either a channel or a channel group.

*callbacktype*

The type of callback. Refer to [FMOD_CHANNELCONTROL_CALLBACK_TYPE](#).

*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

Casting the channel handle
The 'channel' argument can be used when calling functions that are common to Channel and Channelgroup. It can be cast to `FMOD::ChannelControl*` for C++. If a Channel or Channelgroup specific function is needed the 'controltype' argument can be used. The 'controltype' argument is either:

- **FMOD_CHANNELCONTROL_CHANNEL**
  - C++: The 'channel' argument can be cast `FMOD::Channel*`
  - C: The 'channel' argument can be cast to `FMOD_CHANNEL*`

- **FMOD_CHANNELCONTROL_CHANNELGROUP**
  - C++: The 'channel' argument can be cast to `FMOD::ChannelGroup*`
  - C: The 'channel' argument can be cast to `FMOD_CHANNELGROUP*`

'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_CHANNELCONTROL_CALLBACK_END**
  - `commanddata1`: Always 0.
  - `commanddata2`: Always 0.

- **FMOD_CHANNELCONTROL_CALLBACK_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_CHANNELCONTROL_CALLBACK_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_CHANNELCONTROL_CALLBACK_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_CHANNELCONTROL_CALLBACK_TYPE
- System::update
FMOD_CODEC_CLOSE_CALLBACK

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_CLOSE_CALLBACK(
    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_OPEN_CALLBACK`
- `FMOD_CODEC_READ_CALLBACK`
- `FMOD_CODEC_GETLENGTH_CALLBACK`
- `FMOD_CODEC_SETPOSITION_CALLBACK`
- `FMOD_CODEC_GETPOSITION_CALLBACK`
- `FMOD_CODEC_SOUNDCREATE_CALLBACK`
Firelight Technologies FMOD Studio API
**FMOD_CODEC_GETLENGTH_CALLBACK**

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_GETLENGTH_CALLBACK(
    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_OPEN_CALLBACK
- FMOD_CODEC_CLOSE_CALLBACK
- FMOD_CODEC_READ_CALLBACK
- FMOD_CODEC_SETPOSITION_CALLBACK
- FMOD_CODEC_GETPOSITION_CALLBACK
- FMOD_CODEC_SOUNDCREATE_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_CODEC_GETPOSITION_CALLBACK

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_GETPOSITION_CALLBACK(
    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_OPEN_CALLBACK
- FMOD_CODEC_CLOSE_CALLBACK
- FMOD_CODEC_READ_CALLBACK
- FMOD_CODEC_GETLENGTH_CALLBACK
- FMOD_CODEC_SETPOSITION_CALLBACK
- FMOD_CODEC_SOUNDCREATE_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_CODEC_METADATA_CALLBACK

Callback for sounds that have their

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_CODEC_METADATA_CALLBACK(
    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.

`datatype`

Data type of tag. Binary / string / unicode etc. See `FMOD_TAGDATATYPE`.

`unique`

If this is true, then the tag (determined by the name) being updated is the only one of its type. If it is false then there are multiple versions of this tag with the same 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

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_OPEN_CALLBACK
- FMOD_CODEC_CLOSE_CALLBACK
- FMOD_CODEC_READ_CALLBACK
- FMOD_CODEC_GETLENGTH_CALLBACK
- FMOD_CODEC_SETPOSITION_CALLBACK
- FMOD_CODEC_GETPOSITION_CALLBACK
- FMOD_CODEC_SOUNDCREATE_CALLBACK
- FMOD_TAGDATATYPE
Firelight Technologies FMOD Studio API
FMOD_CODEC_OPEN_CALLBACK

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

FMOD_RESULT F_CALLBACK FMOD_CODEC_OPEN_CALLBACK(
    FMOD_CODEC_STATE *codec_state,
    FMOD_MODE usermode,
    FMOD_CREATESOUNDEXINFO *userexinfo
);

Parameters

\textit{codec\_state}

Pointer to the codec state. The user can use this variable to access runtime plugin specific variables and plugin writer user data.

\textit{usermode}

Mode that the user supplied via \texttt{System::createSound}. This is informational and can be ignored, or used if it has relevance to your codec.

\textit{userexinfo}

Extra info structure that the user supplied via \texttt{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 ineeficient 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_CLOSE_CALLBACK`
- `FMOD_CODEC_READ_CALLBACK`
- `FMOD_CODEC_GETLENGTH_CALLBACK`
- `FMOD_CODEC_SETPOSITION_CALLBACK`
- `FMOD_CODEC_GETPOSITION_CALLBACK`
- `FMOD_CODEC_SOUNDCREATE_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_CODEC_READ_CALLBACK

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

```
FMOD_RESULT F_CALLBACK FMOD_CODEC_READ_CALLBACK(
   FMOD_CODEC_STATE *codec_state,
   void *buffer,
   unsigned int samples_in,
   unsigned int *samples_out
);
```
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 \texttt{FMOD_CODEC_WAVEFORMAT}.

samples_in

Number of PCM samples to decode

samples_out

Number of PCM samples decoded
Return Values

If the function succeeds then the return value 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 cannot read number of samples requested, simply return FMOD_OK and give samples_out the number of samples you decoded.

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

- FMOD_CODEC_STATE
- FMOD_CODEC_DESCRIPTION
- FMOD_CODEC_OPEN_CALLBACK
- FMOD_CODEC_CLOSE_CALLBACK
- FMOD_CODEC_GETLENGTH_CALLBACK
- FMOD_CODEC_SETPOSITION_CALLBACK
- FMOD_CODEC_GETPOSITION_CALLBACK
- FMOD_CODEC_SOUNDCREATE_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_CODEC_SETPOSITION_CALLBACK

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_SETPOSITION_CALLBACK(
    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_OPEN_CALLBACK`
- `FMOD_CODEC_CLOSE_CALLBACK`
- `FMOD_CODEC_READ_CALLBACK`
- `FMOD_CODEC_GETLENGTH_CALLBACK`
- `FMOD_CODEC_GETPOSITION_CALLBACK`
- `FMOD_CODEC_SOUNDCREATE_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_CODEC_SOUNDCREATE_CALLBACK

Sound creation callback for the codec when FMOD finishes creating the sound. Ie so the codec can set more parameters for the related created sound, ie loop points/mode or 3D attributes etc.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_CODEC_SOUNDCREATE_CALLBACK(
    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.

`sound`

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 \texttt{FMOD\\_OK} at the bottom of the function, or an appropriate error code from \texttt{FMOD\\_RESULT}.
See Also

- System::createSound
- System::createStream
- FMOD_CODEC_STATE
- FMOD_CODEC_DESCRIPTION
- FMOD_CODEC_OPEN_CALLBACK
- FMOD_CODEC_CLOSE_CALLBACK
- FMOD_CODEC_READ_CALLBACK
- FMOD_CODEC_GETLENGTH_CALLBACK
- FMOD_CODEC_SETPOSITION_CALLBACK
- FMOD_CODEC_GETPOSITION_CALLBACK

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
FMOD_DEBUG_CALLBACK

Callback for debug messages when using the logging version of FMOD.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DEBUG_CALLBACK(
    FMOD_DEBUG_FLAGS flags,
    const char *file,
    int line,
    const char *func,
    const char *message
);
```
**Parameters**

*flags*

Flags which detail the level and type of this log.

*file*

Source code file name where the message originated.

*line*

Source code line number where the message originated.

*func*

Class and function name where the message originated.

*message*

Actual debug message associated with the 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

This callback will fire directly from the log line, as such it can be from any thread.
See Also

- FMOD_DebugInitializer
- FMOD_DEBUG_FLAGS
Firelight Technologies FMOD Studio API
**FMOD_DSP_CREATE_CALLBACK**

This callback is called once when a user creates a DSP unit instance of this type. It is used to allocate memory, initialize variables and the like.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_CREATE_CALLBACK(
    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_RESET_CALLBACK 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`
- `FMOD_DSP_DESCRIPTION`
- `System::createDSP`
- `System::createDSPByType`
- `System::createDSPByPlugin`
- `FMOD_DSP_RESET_CALLBACK`
Firelight Technologies FMOD Studio API
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_DIALOG_CALLBACK(
    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**
- **FMOD_DSP_DESCRIPTION**
- **DSP::showConfigDialog**
Firelight Technologies FMOD Studio API
FMOD_DSP_GETPARAM_BOOL_CALLBACK

This callback is called when the user wants to get an indexed bool parameter from a DSP unit.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_DSP_GETPARAM_BOOL_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    int index,
    FMOD_BOOL *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 bool 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. **Note:** This pointer will be 0 / NULL if a string is not required.
Return Values

If the function succeeds then the return value 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::getParameterBool**.
- **FMOD_DSP_GETPARAM_BOOL_CALLBACK**.

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::getParameterBool`
- `FMOD_DSP_DESCRIPTION`
- `FMOD_DSP_SETPARAM_BOOL_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_DSP_GETPARAM_DATA_CALLBACK

This callback is called when the user wants to get an indexed binary data parameter from a DSP unit.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_GETPARAM_DATA_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    int index,
    void **value,
    unsigned int *length,
    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 void * variable to receive the selected parameter value.

*length*

Pointer to a variable to receive the length of 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. **Note:** This pointer will be 0 / NULL if a string is not required.
Return Values

If the function succeeds then the return value 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::getParameterData**.
- **FMOD_DSP_GETPARAM_DATA_CALLBACK**.

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::getParameterData`
- `FMOD_DSP_DESCRIPTION`
- `FMOD_DSP_SETPARAM_DATA_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_DSP_GETPARAM_FLOAT_CALLBACK

This callback is called when the user wants to get an indexed float parameter from a DSP unit.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_DSP_GETPARAM_FLOAT_CALLBACK(
    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 float 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. **Note:** This pointer will be 0 / NULL if a string is not required.
Return Values

If the function succeeds then the return value 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::getParameterFloat**.
- **FMOD_DSP_GETPARAM_FLOAT_CALLBACK**.

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::getParameterFloat
- FMOD_DSP_DESCRIPTION
- FMOD_DSP_SETPARAM_FLOAT_CALLBACK
Firelight Technologies FMOD Studio API
This callback is called when the user wants to get an indexed int parameter from a DSP unit.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_GETPARAM_INT_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    int index,
    int *value,
    char *valuestr
);
```
Parameters

\textit{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 \texttt{FMOD\_DSP\_STATE} structure.

\textit{index}

The index into the parameter list for the parameter the user wants to get.

\textit{value}

Pointer to a int variable to receive the selected parameter value.

\textit{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. \textbf{Note:} This pointer will be 0 / NULL if a string is not required.
Return Values

If the function succeeds then the return value 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::getParameterInt.
- FMOD_DSP_GETPARAM_INT_CALLBACK.

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::getParameterInt**
- **FMOD_DSP_DESCRIPTION**
- **FMOD_DSP_SETPARAM_INT_CALLBACK**
Firelight Technologies FMOD Studio API
FMOD_DSP_PROCESS_CALLBACK

Alternative callback that replaces FMOD_DSP_READ_CALLBACK and FMOD_DSP_SHOULDIPROCESS_CALLBACK. Can be used to specify the output channel format at runtime rather than create time, and also supports multiple input/output buffers. This callback will be called twice per mix as it has a dual purpose. Once will be with op = FMOD_DSP_PROCESS_QUERY, and then depending on the return value of the query, if it is FMOD_OK it will call it again with FMOD_DSP_PROCESS_PERFORM.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_PROCESS_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    unsigned int length,
    const FMOD_DSP_BUFFER_ARRAY *inbufferarray,
    FMOD_DSP_BUFFER_ARRAY *outbufferarray,
    bool inputsidle,
    FMOD_DSP_PROCESS_OPERATION op
);
```
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.

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.

inbufferarray

Pointer to a FMOD_DSP_BUFFER_ARRAY structure which describes the incoming signal.

outbufferarray

Pointer to a FMOD_DSP_BUFFER_ARRAY structure which describes the outgoing signal.

inputsidle

This is true if no audio is being fed to this unit. Generally used when the FMOD_DSP_PROCESS_OPERATION is set to FMOD_DSP_PROCESS_QUERY, so the FMOD_DSP_PROCESS_PERFORM step can be skipped. Code can then either skip processing, or return silence. If the FMOD_DSP_PROCESS_OPERATION is set to set a countdown timer based on the tail length of their effect and then call FMOD_ERR_DSP_DONTPROCESS, or just immediately return FMOD_ERR_DSP_DONTPROCESS if no further processing is required.

op
Either **FMOD_DSP_PROCESS_QUERY** or **FMOD_DSP_PROCESS_PERFORM**. **FMOD_DSP_PROCESS_QUERY** is only for the purpose of returning **FMOD_OK** or **FMOD_ERR_DSP_DONTPROCESS**. Do not process data in this callback! If op is **FMOD_DSP_PROCESS_PERFORM** then process the input (optionally) and write to the output.
Return Values

If the function succeeds then the return value 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 **FMOD_ERR_DSP_SILENCE** if the effect is generating silence, so FMOD's mixer can optimize the signal path and not process it any more.
See Also

- FMOD_DSP_PROCESS_OPERATION
- FMOD_DSP_READ_CALLBACK
- FMOD_DSP_SHOULDIPROCESS_CALLBACK
- FMOD_DSP_DESCRIPTION
Firelight Technologies FMOD Studio API
**FMOD_DSP_READ_CALLBACK**

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_READ_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    float *inbuffer,
    float *outbuffer,
    unsigned int length,
    int incchannels,
    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.

`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.

`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 **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.

- *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
`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 `FMOD_OK` at the bottom of the function, or an appropriate
error code from `FMOD_RESULT`.

This callback will not be called if the preceeding
`FMOD_DSP_SHOULDIPROCESS_CALLBACK` is returning
`FMOD_ERR_DSP_DONTPROCESS` Return `FMOD_ERR_DSP_SILENCE` if
the effect is generating silence, so FMOD's mixer can optimize the signal path
and not process it any more.

NOTE: Effects that no not stop processing via
`FMOD_DSP_SHOULDIPROCESS_CALLBACK` may keep the signal chain
alive when it is not desirable to do so. FMOD Studio events may return that they
are still playing when they should be stopped.
See Also

- FMOD_DSP_STATE
- FMOD_DSP_SHOULDIPROCESS_CALLBACK
- FMOD_DSP_DESCRIPTION
- DSP::setActive
- FMOD_RESULT
Firelight Technologies FMOD Studio API
FMOD_DSP_RELEASE_CALLBACK

This callback is called when the user releases a DSP unit instance. It is used to free any resources allocated during the course of the lifetime of the DSP instance or perform any shut down code needed to clean up the DSP unit.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_RELEASE_CALLBACK(    
    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 \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.

- **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
- FMOD_DSP_DESCRIPTION
- DSP::release
Firelight Technologies FMOD Studio API
FMOD_DSP_RESET_CALLBACK

This callback function is called by DSP::reset to allow the effect to reset its internal state.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_DSP_RESET_CALLBACK(
    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

This callback is called on all plugins inside an event whenever the event is started (for example by Studio::EventInstance::start).

It is also useful if (for example) an effect is still holding audio data for a sound that has stopped, and is being relocated to a new sound. Resetting the unit would clear any buffers and get it ready for new sound data.

Note that this callback should not change any public parameters that are exposed via FMOD_DSP_DESCRIPTION.paramdesc, but should instead reset the internal state to match the public parameter values.

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
- FMOD_DSP_DESCRIPTION
- DSP::reset
Firelight Technologies FMOD Studio API
**FMOD_DSP_SETPARAM_BOOL_CALLBACK**

This callback is called when the user wants to set a bool parameter for a DSP unit.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_SETPARAM_BOOL_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    int index,
    FMOD_BOOL value
);
```
Parameters

\textit{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 \texttt{FMOD\_DSP\_STATE} structure.

\textit{index}

The index into the parameter list for the parameter the user wants to set.

\textit{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::setParameterBool.

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::setParameterBool
- FMOD_DSP_DESCRIPTION
- FMOD_DSP_GETPARAM_BOOL_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_DSP_SETPARAM_DATA_CALLBACK

This callback is called when the user wants to set a binary data parameter for a DSP unit.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_DSP_SETPARAM_DATA_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    int index,
    void *value,
    unsigned int length
);
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

Pointer to the binary data to set for the selected parameter.

* length

The length of the binary data. 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

Certain data types are predefined by the system and can be specified via the `FMOD_DSP_PARAMETER_DESC_DATA`, see `FMOD_DSP_PARAMETER_DATA_TYPE`.

Functions that the user would have to call for this callback to be called.

- `DSP::setParameterData`.

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::setParameterData
- FMOD_DSP_DESCRIPTION
- FMOD_DSP_GETPARAM_DATA_CALLBACK
Firelight Technologies FMOD Studio API
**FMOD_DSP_SETPARAM_FLOAT_CALLBACK**

This callback is called when the user wants to set a float parameter for a DSP unit.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_SETPARAM_FLOAT_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    int index,
    float value
);
```
Parameters

\textit{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 \texttt{FMOD\_DSP\_STATE} structure.

\textit{index}

The index into the parameter list for the parameter the user wants to set.

\textit{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::setParameterFloat**.

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::setParameterFloat**
- **FMOD_DSP_DESCRIPTION**
- **FMOD_DSP_GETPARAM_FLOAT_CALLBACK**
Firelight Technologies FMOD Studio API
**FMOD_DSP_SETPARAM_INT_CALLBACK**

This callback is called when the user wants to set an int parameter for a DSP unit.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_SETPARAM_INT_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    int index,
    int value
);
```
Parameters

\textit{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 \texttt{FMOD\_DSP\_STATE} structure.

\textit{index}

The index into the parameter list for the parameter the user wants to set.

\textit{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::setParameterInt`.

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::setParameterInt
- FMOD_DSP_DESCRIPTION
- FMOD_DSP_GETPARAM_INT_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_DSP_SETPOSITION_CALLBACK

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_SETPOSITION_CALLBACK(
    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**
- **FMOD_DSP_DESCRIPTION**
- **Channel::setPosition**
Firelight Technologies FMOD Studio API
Called to allow the plugin writer to tell FMOD's mixer whether the
\texttt{FMOD_DSP_READ_CALLBACK} callback should be called or not. This can be
used as an optimization to reduce CPU overhead. If the effect produces silence
such as when it is receiving no signal, then \texttt{FMOD_ERR_DSP_SILENCE} can be
returned in the \texttt{FMOD_DSP_SHOULDIPROCESS_CALLBACK} callback. If
the effect does not modify the sound in any way with the current effect
parameter settings, then \texttt{FMOD_ERR_DSP_DONTPROCESS} can be returned.
Either of these return values will cause FMOD's mixer to skip the
\texttt{FMOD_DSP_READ_CALLBACK} callback.

\textbf{C/C++ Syntax}

\begin{verbatim}
FMOD_RESULT F_CALLBACK FMOD_DSP_SHOULDIPROCESS_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    bool inputsidle,
    unsigned int length,
    FMOD_CHANNELMASK inmask,
    int inchannels,
    FMOD_SPEAKERMODE speakermode
);
\end{verbatim}
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.

inputidle

This is true if no audio is being fed to this unit. Code can then either set a countdown timer based on the tail length of their effect and then call FMOD_ERR_DSP_DONTPROCESS, or just immediately return FMOD_ERR_DSP_DONTPROCESS if no further processing is required.

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.

inmask

A description of the speaker layout of the incoming buffer using FMOD_CHANNELMASK bitfield. For each channel in inchannels parameter, a bit will identify which speaker it belongs to. A value of 0 means default mapping as ordered by FMOD_CHANNELMASK.

inchannels

The number of channels of PCM data in the coming input. A mono signal coming in would be 1. A stereo signal coming in would be 2.

speakermode

A speakermode that corresponds to the channel count and channel mask. Default is FMOD_SPEAKERMODE_DEFAULT. Where it would differ, is if the
channel count is lower than the specified speaker mode's channel count, ie 1 channel could be specified, and use \texttt{FMOD\_SPEAKERMODE\_5POINT1} as the speaker mode, and the mask could tell the callback that it is \texttt{FMOD\_CHANNELMASK\_LOW\_FREQUENCY}.
Return Values

If the function succeeds then the return value 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 example of an effect that would continue processing silence would be an echo or reverb effect that needs to play a tail sound until it fades out to silence. At that point it could return `FMOD_ERR_DSP_SILENCE` as well.

Typically inmask and speakermode parameters will not be important to the plugin, unless it cares about speaker positioning. If it processes any data regardless of channel format coming in, it can safely ignore these 2 parameters.

**NOTE**: Effects that do not stop processing may keep the signal chain alive when it is not desirable to do so. In the case of FMOD Studio it may result in events that keep playing indefinitely.

The following code can be used for DSP effects that have no tail:

```c
static FMOD_RESULT F_CALLBACK shouldIProcess(FMOD_DSP_STATE *dsp_state)
{
    if (inputsidle)
    {
        return FMOD_ERR_DSP_SILENCE;
    }
    return FMOD_OK;
}
```
See Also

- **FMOD_DSP_STATE**
- **FMOD_DSP_READ_CALLBACK**
- **FMOD_DSP_DESCRIPTION**
- **FMOD_RESULT**
Firelight Technologies FMOD Studio API
FMOD_DSP_SYSTEM_DEREGISTER

This callback is called when the user unloads a DSP plugin, or shuts down FMOD using `System::close` or `System::release`. The function can be used as a 'one off' shut down for the plugin after all instances have been freed.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_SYSTEM_DEREGISTER_CALLBACK(
    FMOD_DSP_STATE *dsp_state
);
```
Parameters

\textit{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 instance pointer in the state for this callback will be 0 / null. Only 'systemobject' and 'callbacks' are valid for use.
Return Values

If the function succeeds then the return value 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 callback is not associated with any DSP instance, so the instance member of `FMOD_DSP_STATE` will be 0 / NULL. Functions that the user would have to call for this callback to be called.

- `DSP::release`
- `System::close`
- `System::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`
- `FMOD_DSP_DESCRIPTION`
- `DSP::release`
- `System::close`
- `System::release`
Firelight Technologies FMOD Studio API
This callback is called once, for each mix for this type of DSP. It is not associated with any DSP instance. The function can be used as a global pre/mid/post mix function for this type of DSP to do things like update a global state for the plugin type.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_SYSTEM_MIX_CALLBACK(
    FMOD_DSP_STATE *dsp_state,
    int stage
);
```
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 instance pointer in the state for this callback will be 0 / null. Only 'systemobject' and 'callbacks' are valid for use.

 stage

0 = premix, or before the mixer has executed. 1 = postmix, or after the after the mix has been executed. 2 = midmix, after clocks calculation before the main mix has occurred.
Return Values

If the function succeeds then the return value 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 callback is not associated with any DSP instance, so the instance member of `FMOD_DSP_STATE` will be 0 / NULL. The callback is triggered automatically by the mixer and is not triggered by any API function. Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`. 
See Also

- FMOD_DSP_STATE
- FMOD_DSP_DESCRIPTION
Firelight Technologies FMOD Studio API
FMOD_DSP_SYSTEM_REGISTER_CALLBACK

This callback is called when the user loads a plugin or registers the dsp for the first time, before creating any instances of the effect. The function can be used as a 'one off' init to set up the plugin first, before dsp instances are created.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_DSP_SYSTEM_REGISTER_CALLBACK(
    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 instance pointer in the state for this callback will be 0 / null. Only 'systemobject' and 'callbacks' are valid for use.
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

The callback is not associated with any DSP instance, so the instance member of `FMOD_DSP_STATE` will be 0 / NULL. Functions that the user would have to call for this callback to be called.

- `System::loadPlugin`
- `System::registerDSP`

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`. 
See Also

- `FMOD_DSP_STATE`
- `FMOD_DSP_DESCRIPTION`
- `System::loadPlugin`
- `System::registerDSP`
Firelight Technologies FMOD Studio API
FMOD_FILEASYNCCANCEL_CALLBACK

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

FMOD_RESULT F_CALLBACK FMOD_FILEASYNCCANCEL_CALLBACK(
    FMODASYNCREADINFO *info,
    void *userdata
);
Parameters

info

Pointer to FMOD_ASYNCREADINFO structure. Use this structure to get a reference for your own system, note that the read that use this structure may have already completed.

userdata

Userdata from FMOD_CREATESOUNDEXINFO.fileuserdata or FMOD_STUDIO_BANK_INFO.userData.
Return Values

If the function succeeds then the return value 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_OPEN_CALLBACK
- FMOD_FILE_CLOSE_CALLBACK
- FMOD_FILE_READ_CALLBACK
- FMOD_FILE_SEEK_CALLBACK
- FMOD_FILE_ASYNCREAD_CALLBACK
- FMOD_ASYNCREADINFO
Firelight Technologies FMOD Studio API
**FMOD_FILEASYNCDONE_FUNC**

Function to be called when asynchronous reading is finished.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_ASYNCDONE_FUNC(
    FMODASYNCREADINFO *info,
    FMODRESULT result
);
```
Parameters

info

Pointer to `FMOD_ASYNCREADINFO` structure. Use this structure to get a reference for your own system, note that the read that use this structure may have already completed.

result

The result of the read operation to pass 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

FMOD file system wake up function. Call this when the user file read is finished.
See Also

- FMOD_ASYNCREADINFO
Firelight Technologies FMOD Studio API
Callback for reading from a file asynchronously. Different to \texttt{FMOD\_FILE\_READ\_CALLBACK} 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 \texttt{FMOD\_ASYNCREADINFO} structure (in any thread), and when the \texttt{FMOD\_ASYNCREADINFO::result} field is set to something other than \texttt{FMOD\_ERR\_NOTREADY} then it will continue.

\textbf{C/C++ Syntax}

\begin{verbatim}
FMOD\_RESULT F\_CALLBACK FMOD\_FILE\_ASYNCREAD\_CALLBACK(
    FMOD\_ASYNCREADINFO *info,
    void *userdata
);
\end{verbatim}
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 from FMOD_CREATESOUNDEXINFO.fileuserdata or FMOD_STUDIO_BANK_INFO.userData.
Return Values

If the function succeeds then the return value 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 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.

**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_OPEN_CALLBACK
- FMOD_FILE_CLOSE_CALLBACK
- FMOD_FILE_READ_CALLBACK
- FMOD_FILE_SEEK_CALLBACK
- FMOD_FILEASYNCCANCEL_CALLBACK
- FMOD_ASYNCREADINFO
Firelight Technologies FMOD Studio API
FMOD_FILE_CLOSE_CALLBACK

Callback for closing a file.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_CLOSE_CALLBACK(
    void *handle,
    void *userdata
);
```
Parameters

handle

This is the handle returned from the open callback to use for your own file routines.

userdata

Userdata from FMOD_CREATE_SOUNDEXINFO.fileuserdata or FMOD_STUDIO_BANK_INFO.userData.
Return Values

If the function succeeds then the return value 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_OPEN_CALLBACK
- FMOD_FILE_READ_CALLBACK
- FMOD_FILE_SEEK_CALLBACK
- FMOD_CREATESSOUNDEXINFO
Firelight Technologies FMOD Studio API
FMOD_FILE_OPEN_CALLBACK

Callback for opening a file.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_FILE_OPEN_CALLBACK(
    const char *name,
    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.

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
Userdata from `FMOD_CREATESOUNDEXINFO.fileuserdata` or `FMOD_STUDIO_BANK_INFO.userData`. 
Return Values

If the function succeeds then the return value 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_CLOSE_CALLBACK`
- `FMOD_FILE_READ_CALLBACK`
- `FMOD_FILE_SEEK_CALLBACK`
- `FMOD_CREATE_SOUNDSEXINFO`
Firelight Technologies FMOD Studio API
FMOD_FILE_READ_CALLBACK

Callback for reading from a file.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_FILE_READ_CALLBACK(
    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 from FMOD_CREATESOUNDEXINFO.fileuserdata or FMOD_STUDIO_BANK_INFO.userData.
Return Values

If the function succeeds then the return value 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_OPEN_CALLBACK
- FMOD_FILE_CLOSE_CALLBACK
- FMOD_FILESEEK_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_FILESEEK_CALLBACK

Callback for seeking within a file.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_FILESEEK_CALLBACK(
    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*

Userdata from `FMOD_CREATESOUNDINFO.fileuserdata` or `FMOD_STUDIO_BANK_INFO.userData`. 
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

- **System::setFileSystem**
- **FMOD_FILE_OPEN_CALLBACK**
- **FMOD_FILE_CLOSE_CALLBACK**
- **FMOD_FILE_READ_CALLBACK**
Firelight Technologies FMOD Studio API
Callback to allocate a block of memory.

C/C++ Syntax

```c
void * F_CALLBACK FMOD_MEMORY_ALLOC_CALLBACK(
    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_REALLOC_CALLBACK
- FMOD_MEMORY_FREE_CALLBACK
- FMOD_MEMORY_TYPE
Firelight Technologies FMOD Studio API
Callback to free a block of memory.

**C/C++ Syntax**

```c
void F_CALLBACK FMOD_MEMORY_FREE_CALLBACK(
    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
Firelight Technologies FMOD Studio API
FMOD_MEMORY_REALLOC_CALLBACK

Callback to re-allocate a block of memory to a different size.

C/C++ Syntax

```c
void * F_CALLBACK FMOD_MEMORY_REALLOC_CALLBACK(
    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_ALLOC_CALLBACK
- FMOD_MEMORY_FREE_CALLBACK
- FMOD_MEMORY_TYPE
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_CLOSE_CALLBACK

Shut down callback which is called when the user calls `System::close` or `System::release`. (`System::release` calls `System::close` internally)

**C/C++ Syntax**

```cpp
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_CLOSE_CALLBACK(
    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

- FMOD_OUTPUT_DESCRIPTION
- FMOD_OUTPUT_STATE
- System::release
- System::close

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_GETDRIVERINFO_CALLBACK

Called when the user calls System::getDriverInfo.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_OUTPUT_GETDRIVERINFO_CALLBACK(
    FMOD_OUTPUT_STATE *output_state,
    int id,
    char *name,
    int namelen,
    FMOD_GUID *guid,
    int *systemrate,
    FMOD_SPEAKERMODE *speakermode,
    int *speakermodechannels
);
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_GETNUMDRIVERS_CALLBACK callback.

name

Address of a variable to receive the driver name, encoded as a UTF-8 string, relevant to the index passed in. Fill this in.

namelen

Length of name buffer being passed in by the user.

guid

Pointer to a GUID structure for the user to fill in. A unique identifier here can be used to identify a driver rather than the string.

systemrate

The rate the output device prefers. Leave 0 to remain flexible.

speakermode

The speaker mode the output device prefers. Leave FMOD_SPEAKERMODE_DEFAULT to remain flexible.

speakermodechannels

The speaker mode associated channels the output device prefers. Leave at 0 to remain flexible. More relevant with FMOD_SPEAKERMODE_RAW. This will
be ignored with other speaker modes.
Return Values

If the function succeeds then the return value 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_OUTPUT_DESCRIPTION`
- `FMOD_OUTPUT_STATE`
- `System::getDriverInfo`
- `System::getNumDrivers`
- `FMOD_OUTPUT_GETNUMDRIVERS_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_GETHANDLE_CALLBACK

Called when the user calls `System::getOutputHandle`.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_GETHANDLE_CALLBACK(
    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 receive 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`.
See Also

- FMOD_OUTPUT_DESCRIPTION
- FMOD_OUTPUT_STATE
- System::getOutputHandle
Firelight Technologies FMOD Studio API
Called when the user calls `System::getNumDrivers`.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_GETNUMDRIVERS_CALLBACK(
    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

- **FMOD_OUTPUT_DESCRIPTION**
- **FMOD_OUTPUT_STATE**
- **System::getNumDrivers**
- **System::getDriverInfo**
- **FMOD_OUTPUT_GETDRIVERINFO_CALLBACK**
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_GETPOSITION_C

Returns the current PCM offset or playback position for the output stream. Called from the mixer thread, only when the 'polling' member of FMOD_OUTPUT_DESCRIPTION is set to TRUE.

The internal FMOD output thread calls this function periodically to determine if it should ask for a block of audio data or not.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_GETPOSITION_CALLBACK(
    FMOD_OUTPUT_STATE *output_state,
    unsigned int *pcm
);
```
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.

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.
See Also

- `FMOD_OUTPUT_DESCRIPTION`
- `FMOD_OUTPUT_STATE`
- `FMOD_OUTPUT_LOCK_CALLBACK`
- `FMOD_OUTPUT_UNLOCK_CALLBACK`
Firelight Technologies FMOD Studio API
**FMOD_OUTPUT_INIT_CALLBACK**

Initialization callback which is called when the user calls `System::init`.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_INIT_CALLBACK(
    FMOD_OUTPUT_STATE *output_state,
    int selecteddriver,
    FMOD_INITFLAGS flags,
    int *outputrate,
    FMOD_SPEAKERMODE *speakermode,
    int *speakermodechannels,
    FMOD_SOUND_FORMAT *outputformat,
    int dspbufferlength,
    int dspnumbuffers,
    void *extradriverrdata
);
```
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. FMOD will resample from the rate requested to your rate if they do not match.

speakermode

Speaker mode selected by the user. If not possible, change the speaker mode to the closest match. FMOD will upmix or downmix to the requested speaker mode if they do not match.

speakermodechannels

Speaker mode channel count selected by the user. For example 1 = mono output. 2 = stereo output. Needed if supporting FMOD_SPEAKERMODE_RAW, otherwise it is informational.

outputformat

Sound format supported by output mode. For example, FMOD_SOUND_FORMAT_PCM16 would be normal. FMOD will convert from FMOD_SOUND_FORMAT_PCMFLOAT (the mixer buffer format) to whatever you specify.
**dspbufferlength**

Size of the buffer fmod will mix to in one mix update. This value is in PCM samples.

**dsnumbuffers**

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.

**extradriverrdata**

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 \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

Remember to return **FMOD_OK** at the bottom of the function, or an appropriate error code from **FMOD_RESULT**.
See Also

- **FMOD_OUTPUT_DESCRIPTION**
- **FMOD_OUTPUT_STATE**
- **System::init**
- **System::setDriver**
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_LOCK_CALLBACK

Called from the mixer thread, only when the 'polling' member of FMOD_OUTPUT_DESCRIPTION is set to true.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_OUTPUT_LOCK_CALLBACK(
    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_STATE`
- `FMOD_OUTPUT_UNLOCK_CALLBACK`
- `FMOD_OUTPUT_GETPOSITION_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_MIXER_CALLBACK

Called from the mixer thread, only when the 'polling' member of
`FMOD_OUTPUT_DESCRIPTION` is set to `FALSE`.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_MIXER_CALLBACK(
    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

Called repeatedly to give a thread for waiting on an audio hardware synchronization primitive, used in conjunction with `FMOD_OUTPUT_READFROMMIXER`.

Ensure you have a reasonable timeout (~200ms) on your synchronization primitive and allow this callback to return once per wakeup to avoid deadlocks.

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`. 
See Also

- `FMOD_OUTPUT_DESCRIPTION`
- `FMOD_OUTPUT_STATE`
Firelight Technologies FMOD Studio API
**FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK**

Called from the mixer thread to reserve a hardware resources for a single 3D object. Called during a mix.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK(
    FMOD_OUTPUT_STATE *output_state,
    void **object3d
);
```
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.

object3d

Address of a variable to receive the native 3D audio object pointer, it will be passed into following FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK and FMOD_OUTPUT_OBJECT3DFREE_CALLBACK callbacks.
Return Values

If the function succeeds then the return 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_STATE
- FMOD_OUTPUT_OBJECT3DGETINFO_CALLBACK
- FMOD_OUTPUT_OBJECT3DFREE_CALLBACK
- FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_OBJECT3DFREE_CALLBACK

Called from the mixer thread to release a hardware resource previously acquired with FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK. Called during a mix.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_OUTPUT_OBJECT3DFREE_CALLBACK(
    FMOD_OUTPUT_STATE *output_state,
    void *object3d
);
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.

object3d

Pointer to native 3D audio 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

- FMOD_OUTPUT_DESCRIPTION
- FMOD_OUTPUT_STATE
- FMOD_OUTPUT_OBJECT3DGETINFO_CALLBACK
- FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK
- FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK
Firelight Technologies FMOD Studio API
**FMOD_OUTPUT_OBJECT3DGETINFO_CALLBACK**

Called from the mixer thread to provide information about the capabilities of 3D object hardware. Called during a mix.

### C/C++ Syntax

```c
FMODResultado F_CALLBACK FMOD_OUTPUT_OBJECT3DGETINFO_CALLBACK(
    FMOD_OUTPUT_STATE *output_state,
    int *maxhardwareobjects
);
```
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.

$max\_hardware\_objects$

Maximum number of 3D objects supported by the audio API, if FMOD requires more objects than this number they will be automatically virtualized.
Return Values

If the function succeeds then the return 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_STATE`
- `FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK`
- `FMOD_OUTPUT_OBJECT3DFREE_CALLBACK`
- `FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK

Called from the mixer thread once for every acquired 3D object every mix to provide 3D information and buffered audio. Called during a mix.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK(
    FMOD_OUTPUT_STATE *output_state,
    void *object3d,
    const FMOD_OUTPUT_OBJECT3DINFO *info
);
```
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.

object3d

Pointer to native 3D audio object.

info

Pointer to an information struct that describes the 3D characteristics 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

- FMOD_OUTPUT_DESCRIPTION
- FMOD_OUTPUT_STATE
- FMOD_OUTPUT_OBJECT3DINFO
- FMOD_OUTPUT_OBJECT3DGETINFO_CALLBACK
- FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK
- FMOD_OUTPUT_OBJECT3DFREE_CALLBACK

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_READFROMMIXER

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

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 Studio 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 Studio API
FMOD_OUTPUT_START_CALLBACK

Called just before mixing should begin. Called from System::init.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_OUTPUT_START_CALLBACK(
    FMOD_OUTPUT_STATE *output_state
);

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.
**Return Values**

If the function succeeds then the return value 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_OUTPUT_DESCRIPTION`
- `FMOD_OUTPUT_STATE`
- `FMOD_OUTPUT_STOP_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_STOP_CALLBACK

Called just after mixing has finished. Called from `System::close`.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_OUTPUT_STOP_CALLBACK(
    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

- **FMOD_OUTPUT_DESCRIPTION**
- **FMOD_OUTPUT_STATE**
- **FMOD_OUTPUT_START_CALLBACK**
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_UNLOCK_CALLBACK

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_UNLOCK_CALLBACK(
    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 nescessary and if needed, send it to sound ram.
See Also

- FMOD_OUTPUT_DESCRIPTION
- FMOD_OUTPUT_STATE
- FMOD_OUTPUT_LOCK_CALLBACK
- FMOD_OUTPUT_GETPOSITION_CALLBACK

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_UPDATE_CALLBACK

Called when the user calls System::update.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_OUTPUT_UPDATE_CALLBACK(
    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 \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

Remember to return `FMOD_OK` at the bottom of the function, or an appropriate error code from `FMOD_RESULT`.
See Also

- FMOD_OUTPUT_DESCRIPTION
- FMOD_OUTPUT_STATE
- System::update
Firelight Technologies FMOD Studio API
**FMOD_SOUND_NONBLOCK_CALLBACK**

Callback to be called when a sound has finished loading, or a non blocking seek is occurring.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_SOUND_NONBLOCK_CALLBACK(
    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.

Note that for non-blocking streams a seek could occur when restarting the sound after the first playthrough. This will result in a callback being triggered again.

Since this callback can occur from the async thread, there are restrictions about what functions can be called during the callback. All Sound functions are safe to call, except for `Sound::setSoundGroup` and `Sound::release`. It is also safe to call `System::getUserData`. The rest of the Low Level API and the Studio API is not allowed. Calling a non-allowed function will return `FMOD_ERR_INVALID_THREAD`.
See Also

- System::createSound
- FMOD_CREATESOUNDEXINFO
Firelight Technologies FMOD Studio API
**FMOD_SOUND_PCMREAD_CALLBACK**

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_PCMREAD_CALLBACK(
    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_PCMSETPOS_CALLBACK`
- `System::createSound`
- `System::createStream`
- `FMOD_CREATESOUNDEXINFO`
Firelight Technologies FMOD Studio API
FMOD_SOUND_PCMSETPOS_CALLBACK

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_PCMSETPOS_CALLBACK(
    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), 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.

*postcode*

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_PCMREAD_CALLBACK**
- **System::createSound**
- **System::createStream**
- **FMOD_CREATESOUNDEXINFO**
Firelight Technologies FMOD Studio API
**FMOD_SYSTEM_CALLBACK**

Callback for system events.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_SYSTEM_CALLBACK(
    FMOD_SYSTEM *system,
    FMOD_SYSTEM_CALLBACK_TYPE type,
    void *commanddata1,
    void *commanddata2,
    void *userdata
);
```
Parameters

system
Pointer to a system handle.

type
The type of callback. Refer to FMOD_SYSTEM_CALLBACK_TYPE.

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.

userdata
The userdata assigned into the given system, or NULL if not set. See remarks for more 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.
Remarks

C++ Users. Cast FMOD_SYSTEM * to FMOD::System * inside the callback and use as normal.

'userdata' is the userdata assigned to the system from System::setUserData function. In the case of multiple FMOD systems being created, some callbacks may not have the system context in which case this variable will be last userdata set into any system.

'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_CALLBACK_DEVICELISTCHANGED**
  - \texttt{commanddata1}: Always 0.
  - \texttt{commanddata2}: Always 0.

- **FMOD_SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED**
  - \texttt{commanddata1}: A string (char*) which represents the file and line number of the allocation inside FMOD.
  - \texttt{commanddata2}: The size (int) of the requested allocation.

- **FMOD_SYSTEM_CALLBACK_THREADCREATED**
  - \texttt{commanddata1}: The handle of the created thread. See notes below for thread handle types
  - \texttt{commanddata2}: A string (char*) which represents the name of the thread.

- **FMOD_SYSTEM_CALLBACK_THREADDESTROYED**
  - \texttt{commanddata1}: The handle of the destroyed thread. See notes below for thread handle types
  - \texttt{commanddata2}: A string (char*) which represents the name of the thread.

- **FMOD_SYSTEM_CALLBACK_BADDSPCONNECTION**
  - \texttt{commanddata1}: Pointer to a FMOD::DSP object that was the target of the DSP connection.
  - \texttt{commanddata2}: Pointer to a FMOD::DSP object that was the source of the DSP connection.

- **FMOD_SYSTEM_CALLBACK_PREMIX**
  - \texttt{commanddata1}: 0.
  - \texttt{commanddata2}: 0.

- **FMOD_SYSTEM_CALLBACK_MIDMIX**
commanddata1: 0.
commanddata2: 0.

- **FMOD_SYSTEM_CALLBACK_POSTMIX**
  commanddata1: 0.
  commanddata2: 0.

- **FMOD_SYSTEM_CALLBACK_ERROR**
  commanddata1: Pointer to a `FMOD_ERRORCALLBACK_INFO` structure with extra information about the error.
  commanddata2: 0.

**Note!** For `FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED`, the user must call `System::update` for the callback to trigger! See `FMOD_SYSTEM_CALLBACK_TYPE` for details.

**Note!** For `FMOD_SYSTEM_CALLBACK_THREADCREATED` and `FMOD_SYSTEM_CALLBACK_THREADDESTROYED`, the handle that is returned (via commanddata1) is different on each platform. The types to cast to are as follows.

- Mac, Linux, iOS, Android: pthread_t
- PS3: sys_ppu_thread_t
- PS4: Scepthread
- WiiU: OSThread
- Win, Xbox360, XboxOne: HANDLE
- WinStore: IAsyncAction
- PSVita: FMOD_OS_Thread
  This is a custom struct you can define as:
  ```c
  typedef struct FMOD_OS_Thread {
      SceUID id;
      int (*func)(void *param);
      void *param;
  };
  ```

Here is an example of a system callback.

```c
FMOD_RESULT F_CALLBACK systemcallback(FMOD_SYSTEM *system, FMOD_SYSTEM_CALLBACK_TYPE *type)
{
    FMOD::System *sys = (FMOD::System *)system;

    switch (type)
    {
        case FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED:
        {
            int numdrivers;

            printf("NOTE : FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED oc");
        }
        ```
sys->getNumDrivers(&numdrivers);

printf("Numdevices = %d\n", numdrivers);
break;
}

case FMOD_SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED:
{
    printf("ERROR : FMOD_SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED\n" unt, commanddata1);
    printf("%d bytes.\n", commanddata2);
    break;
}

case FMOD_SYSTEM_CALLBACK_THREADCREATED:
{
    printf("NOTE : FMOD_SYSTEM_CALLBACK_THREADCREATED occurred.\n") unt, commanddata1);
    printf("Thread ID = %d\n", (int)commanddata1);
    printf("Thread Name = %s\n", (char *)commanddata2);
    break;
}

case FMOD_SYSTEM_CALLBACK_THREADDESTROYED:
{
    printf("NOTE : FMOD_SYSTEM_CALLBACK_THREADDESTROYED occurred.\n") unt, commanddata1);
    printf("Thread ID = %d\n", (int)commanddata1);
    printf("Thread Name = %s\n", (char *)commanddata2);
    break;
}

case FMOD_SYSTEM_CALLBACK_BADDSPCONNECTION:
{
    FMOD::DSP *source = (FMOD::DSP *)commanddata1;
    FMOD::DSP *dest = (FMOD::DSP *)commanddata2;

    printf("ERROR : FMOD_SYSTEM_CALLBACK_BADDSPCONNECTION occurred.\n")
    if (source) {
        char name[256];
        source->getInfo(name, 0, 0, 0);
        printf("SOURCE = %s\n", name);
    }
    if (dest) {
        char name[256];
        dest->getInfo(name, 0, 0, 0);
        printf("DEST = %s\n", name);
    }
    break;
}

case FMOD_SYSTEM_CALLBACK_PREMIX:
{
    printf("NOTE : FMOD_SYSTEM_CALLBACK_PREMIX occurred.\n")
break;
}
case FMOD_SYSTEM_CALLBACK_MIDMIX:
{
printf("NOTE : FMOD_SYSTEM_CALLBACK_MIDMIX occurred.\n")
break;
}
case FMOD_SYSTEM_CALLBACK_POSTMIX:
{
printf("NOTE : FMOD_SYSTEM_CALLBACK_POSTMIX occurred.\n")
break;
}
case FMOD_SYSTEM_CALLBACK_ERROR:
{
    FMOD_ERRORCALLBACK_INFO* info = (FMOD_ERRORCALLBACK_INFO
printf("NOTE : FMOD_SYSTEM_CALLBACK_ERROR occurred.\n");
printf(" ERROR (%d) from %s(%s) with instance %p (type %d)\n",
break;
}
return FMOD_OK;
}
See Also

- System::setCallback
- System::setUserData
- FMOD_SYSTEM_CALLBACK_TYPE
- System::update
Firelight Technologies FMOD Studio API
Structures

FMOD_3D_ATTRIBUTES FMOD_ADVANCEDSETTINGS
FMOD_ASYNCREADINFO
FMOD_CODEC_DESCRIPTION
FMOD_CODEC_STATE
FMOD_CODEC_WAVEFORMAT
FMOD_COMPLEX
FMOD_CREATESOUNDEXINFO
FMOD_DSP_BUFFER_ARRAY
FMOD_DSP_DESCRIPTION
FMOD_DSP_METERING_INFO
FMOD_DSP_PARAMETER_3DATTRIBUTES
FMOD_DSP_PARAMETER_3DATTRIBUTES_MULTI
FMOD_DSP_PARAMETER_DESC
FMOD_DSP_PARAMETER_DESC_BOOL
FMOD_DSP_PARAMETER_DESC_DATA
FMOD_DSP_PARAMETER_DESC_FLOAT
FMOD_DSP_PARAMETER_DESC_INT
FMOD_DSP_PARAMETER_FFT
FMOD_DSP_PARAMETER_FLOAT_MAPPING
FMOD_DSP_PARAMETER_FLOAT_MAPPING_PIECEWISE_LINEAR
FMOD_DSP_PARAMETER_OVERALLGAIN
FMOD_DSP_PARAMETER_SIDECHAIN
FMOD_DSP_STATE
FMOD_DSP_STATE_DFT_FUNCTIONS
FMOD_DSP_STATE_FUNCTIONS
FMOD_DSP_STATE_PAN_FUNCTIONS
FMOD_ERROR_CALLBACK_INFO
FMOD_GUID
FMOD_OUTPUT_DESCRIPTION
FMOD_OUTPUT_OBJECT3DINFO
FMOD_OUTPUT_STATE
FMOD_PLUGINLIST
FMOD_REVERB_PROPERTIES
FMOD_TAG
FMOD_VECTOR
Firelight Technologies FMOD Studio API
**FMOD_3D_ATTRIBUTES**

Structure describing a position, velocity and orientation.

**C/C++ Syntax**

```c
typedef struct {
    FMOD_VECTOR position;
    FMOD_VECTOR velocity;
    FMOD_VECTOR forward;
    FMOD_VECTOR up;
} FMOD_3D_ATTRIBUTES;
```
Members

*position*

The position of the object in world space, measured in distance units.

*velocity*

The velocity of the object measured in distance units **per second**.

*forward*

The forwards orientation of the object. This vector must be of unit length (1.0) and perpendicular to the up vector.

*up*

The upwards orientation of the object. This vector must be of unit length (1.0) and perpendicular to the forward vector.
**Remarks**

Attributes should use your chosen coordinate system, see [3D sounds](#) for more information.

**JavaScript only:**

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD._3D_ATTRIBUTES()", no 'new' keyword is required.
See Also

- FMOD_VECTOR
- FMOD_DSP_PARAMETER_3DATTRIBUTES
Firelight Technologies FMOD Studio API
FMOD_ADVANCEDSETTINGS

Settings for advanced features like configuring memory and cpu usage for the FMOD_CREATECOMPRESSESAMPLE feature.

C/C++ Syntax

typedef struct {
    int cbSize;
    int maxMPEGCodecs;
    int maxADPCMCodecs;
    int maxXMACodecs;
    int maxVorbisCodecs;
    int maxAT9Codecs;
    int maxFADPCMCodexes;
    int maxPCMCodexes;
    int ASIONumChannels;
    char **ASIOChannelList;
    FMOD_Speaker *ASIOSpeakerList;
    float HRTFMInAngle;
    float HRTFMaxAngle;
    float HRTFFreq;
    float vol0virtualvol;
    unsigned int defaultDecodeBufferSize;
    unsigned short profilePort;
    unsigned int geometryMaxFadeTime;
    float distanceFilterCenterFreq;
    int reverb3Dinstance;
    int DSPBufferPoolSize;
    unsigned int stackSizeStream;
    unsigned int stackSizeNonBlocking;
    unsigned int stackSizeMixer;
    FMOD_DSP_Resampler resamplerMethod;
    unsigned int commandQueueSize;
    unsigned int randomSeed;
} FMOD_ADVANCEDSETTINGS;

JavaScript Syntax

struct FMOD_ADVANCEDSETTINGS
{
    maxMPEGCodecs,
    maxADPCMCodecs,
}
maxXMACodecs,
maxVorbisCodecs,
maxAT9Codecs,
maxFADPCMCodes,
maxPCMCodecs,
ASIONumChannels,
HRTFMinAngle,
HRTFMaxAngle,
HRTFFreq,
vol0virtualvol,
defaultDecodeBufferSize,
profilePort,
geometryMaxFadeTime,
distanceFilterCenterFreq,
reverb3Dinstance,
DSPBufferPoolSize,
stackSizeStream,
stackSizeNonBlocking,
stackSizeMixer,
resamplerMethod,
commandQueueSize,
randomSeed,
};
Members

\textit{cbSize}

[w] Size of this structure. Use \texttt{sizeof(FMOD\_ADVANCED\_SETTINGS)} \textbf{NOTE:} This must be set before calling \texttt{System::getAdvancedSettings} or \texttt{System::setAdvancedSettings}!

\textit{maxMPEGCodecs}

\texttt{[r/w]} Optional. Specify 0 to ignore. For use with \texttt{FMOD\_CREATE\_COMPRESSED\_SAMPLE} only. MPEG codecs consume 22,216 bytes per instance and this number will determine how many MPEG channels can be played simultaneously. Default = 32.

\textit{maxADPCMCodecs}

\texttt{[r/w]} Optional. Specify 0 to ignore. For use with \texttt{FMOD\_CREATE\_COMPRESSED\_SAMPLE} only. ADPCM codecs consume 2,480 bytes per instance and this number will determine how many ADPCM channels can be played simultaneously. Default = 32.

\textit{maxXMACodecs}

\texttt{[r/w]} Optional. Specify 0 to ignore. For use with \texttt{FMOD\_CREATE\_COMPRESSED\_SAMPLE} only. XMA codecs consume 6,263 bytes per instance and this number will determine how many XMA channels can be played simultaneously. Default = 32.

\textit{maxVorbisCodecs}

\texttt{[r/w]} Optional. Specify 0 to ignore. For use with \texttt{FMOD\_CREATE\_COMPRESSED\_SAMPLE} only. Vorbis codecs consume 16,512 bytes per instance and this number will determine how many Vorbis channels can be played simultaneously. Default = 32.

\textit{maxAT9Codecs}

\texttt{[r/w]} Optional. Specify 0 to ignore. For use with
**FMOD_CREATECOMPRESSEDSAMPLE** only. AT9 codecs consume 20,664 bytes per instance and this number will determine how many AT9 channels can be played simultaneously. Default = 32.

**maxFADPCMCODECS**

[r/w] Optional. Specify 0 to ignore. For use with **FMOD_CREATECOMPRESSEDSAMPLE** only. FADPCM codecs consume 2,232 bytes per instance and this number will determine how many FADPCM channels can be played simultaneously. Default = 32.

**maxPCMCodecs**

[r/w] Optional. Specify 0 to ignore. For use with PS3 only. PCM codecs consume 2,536 bytes per instance and this number will determine how many streams and PCM voices can be played simultaneously. Default = 32.

**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.

**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 = 4000.0.

**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 below this, then the sound will become virtual. Use this value to raise the threshold to a different point where a sound goes virtual.

**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

**profilePort**

[r/w] Optional. Specify 0 to ignore. For use with FMOD_INIT_PROFILE_ENABLE. 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.

**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.

reverb3Dinstance

[r/w] Optional. Specify 0 to ignore. Out of 0 to 3, 3d reverb spheres will create a physical reverb unit on this instance slot. See FMOD_REVERB_PROPERTIES.

DSPBufferPoolSize

[r/w] Optional. Specify 0 to ignore. Number of buffers in DSP buffer pool. Each buffer will be DSPBlockSize * sizeof(float) * SpeakerModeChannelCount. ie 7.1 @ 1024 DSP block size = 8 * 1024 * 4 = 32kb. Default = 8.

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)

resamplerMethod

[r/w] Optional. Specify 0 to ignore. Resampling method used with fmod's software mixer. See FMOD_DSP_RESAMPLER for details on methods.

commandQueueSize

[r/w] Optional. Specify 0 to ignore. Specify the command queue size for thread safe processing. Default 2048 (2kb)
randomSeed

[r/w] Optional. Specify 0 to ignore. Seed value that FMOD will use to initialize its internal random number generators.
Remarks

maxMPEGCodecs / maxADPCMCodecs / 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_CREATECOMPRESSESAMPLE 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).

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.ADVANCEDSETTINGS()", no 'new' keyword is required.
See Also

- System::setAdvancedSettings
- System::getAdvancedSettings
- System::init
- FMOD_MODE
Firelight Technologies FMOD Studio API
**FMODASYNCREADINFO**

Structure that is passed into **FMOD_FILEASYNCREAD_CALLBACK**. Use the information in this structure to perform

**C/C++ Syntax**

```c
typedef struct {
    void *handle;
    unsigned int offset;
    unsigned int sizebytes;
    int priority;
    void *userdata;
    void *buffer;
    unsigned int bytesread;
    FMOD_FILEASYNCDONE_FUNC done;
} FMODASYNCREADINFO;
```

**JavaScript Syntax**

```javascript
struct FMODASYNCREADINFO {
    handle,
    offset,
    sizebytes,
    priority,
    userdata,
    buffer,
    bytesread,
    done,
};
```
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')

userdata
[r/w] User data pointer specific to this request. Initially 0, can be ignored or set by the user. Not related to the file's main userdata member.

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.

done
[r] FMOD file system wake up function. Call this when user file read is finished. Pass result of file read as a parameter.
**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** calling 'done'. As soon as done is called, FMOD will asynchronously continue internally using the data provided in this structure.

Set result in the 'done' function pointer 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**.

**JavaScript only :**

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.ASYNCREADINFO()", no 'new' keyword is required.
See Also

- FMOD_FILEASYNCREAD_CALLBACK
- FMOD_FILEASYNCCANCEL_CALLBACK
- FMOD_FILEASYNCDONE_FUNC
Firelight Technologies FMOD Studio API
FMOD_CODEC_DESCRIPTION

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.

C/C++ Syntax

typedef struct {
    const char *name;
    unsigned int version;
    int defaultasstream;
    FMOD_TIMEUNIT timeunits;
    FMOD_CODEC_OPEN_CALLBACK open;
    FMOD_CODEC_CLOSE_CALLBACK close;
    FMOD_CODEC_READ_CALLBACK read;
    FMOD_CODEC_GETLENGTH_CALLBACK getlength;
    FMOD_CODEC_SETPOSITION_CALLBACK setposition;
    FMOD_CODEC_GETPOSITION_CALLBACK getposition;
    FMOD_CODEC_SOUNDCREATE_CALLBACK soundcreate;
    FMOD_CODEC_GETWAVEFORMAT_CALLBACK getwaveformat;
} FMOD_CODEC_DESCRIPTION;

JavaScript Syntax

struct FMOD_CODEC_DESCRIPTION
{
    name,
    version,
    defaultasstream,
    timeunits,
    open,
    close,
    read,
    getlength,
    setposition,
    getposition,
    soundcreate,
    getwaveformat,
};
Members

name

[w] Name of the codec.

version

[w] Plugin writer's version number.

defaultasstream

[w] 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

[w] When setposition codec is called, only these time formats will be passed to the codec. Use bitwise OR to accumulate different types.

open

[w] Open callback for the codec for when FMOD tries to open a sound using this codec.

close

[w] Close callback for the codec for when FMOD tries to close a sound using this codec.

read
[w] 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

[w] Callback to return the length of the song in whatever format required when
Sound::getLength is called.

setposition

[w] Seek callback for the codec for when FMOD tries to seek within the file
with Channel::setPosition.

getposition

[w] Tell callback for the codec for when FMOD tries to get the current position
within the with Channel::getPosition.

soundcreate

[w] 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

[w] 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 [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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time.
To initialize an new instance in javascript use "FMOD.CODEC_DESCRIPTION()", no 'new' keyword is required.
See Also

- FMOD_CODEC_STATE
- FMOD_CODEC_WAVEFORMAT
Firelight Technologies FMOD Studio API
FMOD_CODEC_STATE

Codec plugin structure that is passed into each callback.

Optionally set the numsubsounds and waveformat members when called in \texttt{FMOD_CODEC_OPEN_CALLBACK} to tell fmod what sort of sound to create.

\textbf{C/C++ Syntax}

\begin{verbatim}
typedef struct {
    int numsubsounds;
    FMOD_CODEC_WAVEFORMAT *waveformat;
    void *plugindata;
    void *filehandle;
    unsigned int filesize;
    FMOD_FILE_READ_CALLBACK fileread;
    FMOD_FILE_SEEK_CALLBACK fileseek;
    FMOD_CODEC_METADATA_CALLBACK metadata;
    int waveformatversion;
} FMOD_CODEC_STATE;
\end{verbatim}

\textbf{JavaScript Syntax}

\begin{verbatim}
struct FMOD_CODEC_STATE
{
    numsubsounds,
    plugindata,
    filehandle,
    filesize,
    fileread,
    fileseek,
    metadata,
    waveformatversion,
};
\end{verbatim}
Members

numsubsounds

[w] Number of 'subsounds' in this sound. Anything other than 0 makes it a 'container' format (ie 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

[w] Pointer to an array of format structures containing information about each sample. Can be 0 or NULL if FMOD_CODEC_GETWAVEFORMAT_CALLBACK 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

[w] Plugin writer created data the codec author wants to attach to this object.

filehandle

[r] This will return an internal FMOD file handle to use with the callbacks provided.

filesize

[r] This will contain the size of the file in bytes.

fileread

[r] This will return a callable FMOD file function to use from codec.

fileseek

[r] This will return a callable FMOD file function to use from codec.
metadata

[r] This will return a callable FMOD metadata function to use from codec.

waveformatversion

[w] Must be set to FMOD_CODEC_WAVEFORMAT_VERSION in the FMOD_CODEC_OPEN_CALLBACK.
**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.

'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 FSB (contains multiple sounds), DLS (contain instruments).

The waveform format value should point to an 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 waveform, the same as if the number of subsounds was 1. If subsounds was 100 for example, there should be a pointer to an array of 100 waveform structures.

The waveform pointer is optional and could be 0, if using FMOD_CODEC_GETWAVEFORMAT_CALLBACK is preferred.

When a sound has 1 or more subsounds, the caller must play the individual sounds specified by first obtaining the subsound with `Sound::getSubSound`.

**JavaScript only :**

Not all fields are currently supported or may not work as expected at this time. To initialize a new instance in javascript use "FMOD.CODEC_STATE()", no 'new' keyword is required.
See Also

- FMOD_CODEC_WAVEFORMAT
- FMOD_FILE_READ_CALLBACK
- FMOD_FILE_SEEK_CALLBACK
- FMOD_CODEC_METADATA_CALLBACK
- Sound::getSubSound
- Sound::getNumSubSounds
- FMOD_CODEC_WAVEFORMAT_VERSION
Firelight Technologies FMOD Studio API
FMOD_CODEC_WAVEFORMAT

Set these values marked to tell fmod what sort of sound to create when the codec open callback is called.
The format, channels, frequency and lengthpcm tell FMOD what sort of sound buffer to create when you initialize your code.
If you wrote an MP3 codec that decoded to stereo 16bit integer PCM for a 44khz sound, you would specify FMOD_SOUND_FORMAT_PCM16, and channels would be equal to 2, and frequency would be 44100.

C/C++ Syntax

typedef struct {
    const char* name;
    FMOD_SOUND_FORMAT format;
    int channels;
    int frequency;
    unsigned int lengthbytes;
    unsigned int lengthpcm;
    unsigned int pcmblocksize;
    int loopstart;
    int loopend;
    FMOD_MODE mode;
    FMOD_CHANNELMASK channelmask;
    FMOD_CHANNELORDER channelorder;
    float peakvolume;
} FMOD_CODEC_WAVEFORMAT;

JavaScript Syntax

struct FMOD_CODEC_WAVEFORMAT {
    name,
    format,
    channels,
    frequency,
    lengthbytes,
    lengthpcm,
    pcmblocksize,
    loopstart,
    loopend,
    mode,
channelmask,
channelorder,
peakvolume,
};
Members

name

[w] Name of sound. Optional. If used, the codec must own the lifetime of the string memory until the codec is destroyed.

format

[w] Format for (decompressed) codec output, ie
FMOD_SOUND_FORMAT_PCM8, FMOD_SOUND_FORMAT_PCM16.
Mandatory - Must be supplied.

channels

[w] Number of channels used by codec, ie mono = 1, stereo = 2. Mandatory - Must be supplied.

frequency

[w] Default frequency in hz of the codec, ie 44100. Mandatory - Must be supplied.

lengthbytes

[w] Length in bytes of the source data. Used for
FMOD_TIMEUNIT_RAWBYTES. Optional. Default = 0.

lengthpcm

[w] 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. Mandatory - Must be supplied.

pcmblocksize

[w] Minimum, optimal number of decompressed PCM samples codec can handle. 0 or 1 = no buffering. Anything higher means FMOD will allocate a PCM buffer of this size to read in chunks. The codec read callback will be called
in multiples of this value. Optional.

**loopstart**

[w] Loopstart in decompressed, PCM samples of file. Optional. Default = 0.

**loopend**

[w] Loopend in decompressed, PCM samples of file. Optional. Default = 0.

**mode**

[w] Mode to determine whether the sound should by default load as looping, nonlooping, 2d or 3d. Optional. Default = **FMOD_DEFAULT**.

**channelmask**

[w] Defined channel bitmask to describe which speakers the channels in the codec map to, in order of channel count. See fmod_common.h. Optional. Leave at 0 to map to the speaker layout defined in **FMOD_SPEAKER**.

**channelorder**

[w] Defined channel order type, to describe where each sound channel should pan for the number of channels specified. See fmod_common.h. Optional. Leave at 0 to play in default speaker order.

**peakvolume**

[w] Peak volume of sound. Optional. Default = 0 if not used.
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.

1.07 Note. 'blockalign' member which was in bytes has been removed. 'pcmblocksize' is now the replacement, and is measured in PCM samples only, not bytes. This is purely to support buffering internal to FMOD for codecs that are not sample accurate.
Note: When registering a codec, format, channels, frequency and lengthpcm must be supplied, otherwise there will be an error.
This structure is optional if FMOD_CODEC_GETWAVEFORMAT_CALLBACK is specified.
An array of these structures may be needed if FMOD_CODEC_STATE::numsubsounds is larger than 1.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.CODEC_WAVEFORMAT()", no 'new' keyword is required.
See Also

- **FMOD_CODEC_STATE**
- **FMOD_SOUND_FORMAT**
- **FMOD_MODE**
- **FMOD_CHANNELMASK**
- **FMOD_CHANNELORDER**
- **FMOD_CODEC_WAVEFORMAT_VERSION**
Firelight Technologies FMOD Studio API
FMOD_COMPLEX

Complex number structure used for holding FFT frequency domain-data for FMOD_FFTREAL and FMOD_IFFTREAL DSP functions.

C/C++ Syntax

typedef struct {
    float real;
    float imag;
} FMOD_COMPLEX;

JavaScript Syntax

struct FMOD_COMPLEX {
    real,
    imag,
};
Members

real
Real component

imag
Imaginary component
Remarks

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.COMPLEX()", no 'new' keyword is required.
See Also

- FMODE_DSP_STATE_FUNCTIONS
- FMODE_DSP_STATE_DFT_FUNCTIONS
Firelight Technologies FMOD Studio API
FMOD_CREATESOUNDEXINFO

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) when created as a stream.
- To specify which subsounds to load for multi-sample sounds (ie FSB/DLS) 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 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.

C/C++ Syntax

typedef struct {
    int cbsize;
    unsigned int length;
    unsigned int fileoffset;
    int numchannels;
    int defaultfrequency;
    MOD_SOUND_FORMAT format;
    unsigned int decodebuffersize;
    int initialsounds;
    int numsubsounds;
    int *inclusionlist;
    int inclusionlistnum;
    MOD_SOUND_PCMREAD_CALLBACK pcmreadcallback;
    MOD_SOUND_PCMSETPOS_CALLBACK pcmsetposcallback;
    MOD_SOUND_NONBLOCK_CALLBACK nonblockcallback;
    const char *dlsname;
    const char *encryptionkey;
}
int maxpolyphony;
void *userdata;
FMOD_SOUND_TYPE suggestedsoundtype;
FMOD_FILE_OPEN_CALLBACK fileuseropen;
FMOD_FILE_CLOSE_CALLBACK fileuserclose;
FMOD_FILE_READ_CALLBACK fileuserread;
FMOD_FILESEEK_CALLBACK fileuserseek;
FMOD_FILEASYNCREAD_CALLBACK fileusersyncread;
FMOD_FILEASYNCCANCEL_CALLBACK fileusersynccancel;
void *fileuserdata;
int filebuffersize;
FMOD_CHANNELORDER channelorder;
FMOD_CHANNELMASK channelmask;
FMOD_SOUNDGROUP *initialsoundgroup;
unsigned int initialseekposition;
FMOD_TIMEUNIT initialseekpostype;
int ignoresetfilesystem;
unsigned int audioqueuepolicy;
unsigned int minmidigranularity;
int nonblockthreadid;
FMOD_GUID *fsbguid;
} FMOD_CREATESOUNDEXINFO;

JavaScript Syntax

struct FMOD_CREATESOUNDEXINFO
{
    length,
    fileoffset,
    numchannels,
    defaultfrequency,
    format,
    decodebuffersize,
    initialsubsound,
    numsubsounds,
    inclusionlistnum,
    pcmreadcallback,
    pcmsetposcallback,
    nonblockcallback,
    dlsname,
    encryptionkey,
    maxpolyphony,
    userdata,
    suggestedsoundtype,
    fileuseropen,
    fileuserclose,
    fileuserread,
    fileuserseek,
    fileusersyncread,
fileuserasynccancel,
fileuserdata,
filebuffersize,
channelorder,
channelmask,
initialsoundgroup,
initialseekposition,
initialseekpostype,
ignoreresetfilesystem,
audioqueuepolicy,
minmidigranularity,
nonblockthreadid,
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 Studio.

length

[w] Optional. Specify 0 to ignore. Number of bytes to load starting at 'fileoffset', or size of sound to create (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. Can be specified up to FMOD_MAX_CHANNEL_WIDTH.

defaultfrequency

[w] Optional. Specify 0 to ignore. Default frequency of sound in Hz, 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.
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

Optional. Specify 0 to ignore. In a multi-sample file format such as .FSB/.DLS, specify the initial subsound to seek to, only if FMOD_CREATESTREAM is used.

numsubsounds

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

Optional. Specify 0 to ignore. In a multi-sample format such as .FSB/.DLS 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. A single subsound index can be encoded in-place by setting inclusionlistnum to 0, setting the low bit of inclusionlist to 1 and OR the index into inclusionlist shifted left by 1 (optional advanced technique to avoid pointing to additional memory).

inclusionlistnum

Optional. Specify 0 to ignore. This is the number of integers contained within the inclusionlist array.

pcmreadcallback

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. Also called during seeking, when setPosition is called or a stream is restarted.

**dlsname**

[w] Optional. Specify 0 to ignore. Filename for a DLS 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**

[w] Optional. Specify 0 to ignore. Key for encrypted FSB file. Without this key an encrypted FSB file will not load.

**maxpolyphony**

[w] 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**

[w] 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_OPEN_CALLBACK** - use fileuserdata for that.

**suggestedsoundtype**

[w] 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.

**fileuseropen**


**fileuserclose**


**fileuserread**

[w] Optional. Specify 0 to ignore. Callback for reading from this file.

**fileuserseek**

[w] Optional. Specify 0 to ignore. Callback for seeking within this file.

**fileuserasyncread**

[w] Optional. Specify 0 to ignore. Callback for seeking within this file.

**fileuserasynccancel**

[w] Optional. Specify 0 to ignore. Callback for seeking within this file.

**fileuserdata**

[w] Optional. Specify 0 to ignore. User data to be passed into the file callbacks.

**filebuffersize**

[w] Optional. Specify 0 to ignore. Buffer size for reading the file, -1 to disable buffering, or 0 for system default.
channelorder

[w] Optional. Specify 0 to ignore. Use this to differ the way fmod maps multichannel sounds to speakers. See FMOD_CHANNELORDER for more.

channelmask

[w] Optional. Specify 0 to ignore. Use this to specify which channels map to which speakers. See FMOD_CHANNELMASK 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.

ignoresetfilesystem

[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).

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.

fsbguid

[r/w] Optional. Specify 0 to ignore. Allows you to provide the GUID lookup for cached FSB header info. Once loaded the GUID will be written back to the pointer. This is to avoid seeking and reading the FSB header.
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_OPENUSER` 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).**

- Mandatory. Specify 'initialsubsound'.

**To specify which subsounds to load for multi-sample sounds (ie FSB/DLS) 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 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 (i.e., 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 numsubsounds 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, i.e., no null subsounds. As an example, if there are 10,000 subsounds and there is an inclusionlist with only 1 entry, and numsubsounds = 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.

**JavaScript only:**

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.CREATESOUNDEXINFO()", no 'new' keyword is required.
See Also

- System::createSound
- System::setStreamBufferSize
- FMOD_MODE
- FMOD_SOUND_FORMAT
- FMOD_SOUND_TYPE
- FMOD_CHANNELMASK
- FMOD_CHANNELORDER
- FMOD_MAX_CHANNEL_WIDTH
Firelight Technologies FMOD Studio API
FMOD_DSP_BUFFER_ARRAY

Structure for FMOD_DSP_PROCESS_CALLBACK input and output buffers.

C/C++ Syntax

typedef struct {
    int numbuffers;
    int *buffernumchannels;
    FMOD_CHANNELMASK *bufferchannelmask;
    float **buffers;
    FMOD_SPEAKERMODE speakermode;
} FMOD_DSP_BUFFER_ARRAY;

JavaScript Syntax

struct FMOD_DSP_BUFFER_ARRAY
{
    numbuffers,
    speakermode,
};
Members

numbuffers
[r/w] number of buffers

buffernumchannels
[r/w] array of number of channels for each buffer

bufferchannelmask
[r/w] array of channel masks for each buffer

buffers
[r/w] array of buffers

speakermode
[r/w] speaker mode for all buffers in the array
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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time.
To initialize an new instance in javascript use
"FMOD.DSP BUFFER ARRAY()", no 'new' keyword is required.
See Also

- FMOD_DSP_DESCRIPTION
Firelight Technologies FMOD Studio API
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.

C/C++ Syntax

typedef struct {
  unsigned int pluginsdkversion;
  char name[32];
  unsigned int version;
  int numinputbuffers;
  int numoutputbuffers;
  FMOD_DSP_CREATE_CALLBACK create;
  FMOD_DSP_RELEASE_CALLBACK release;
  FMOD_DSP_RESET_CALLBACK reset;
  FMOD_DSP_READ_CALLBACK read;
  FMOD_DSP_PROCESS_CALLBACK process;
  FMOD_DSP_SETPOSITION_CALLBACK setposition;
  int numparameters;
  FMOD_DSP_PARAMETER_DESC **paramdesc;
  FMOD_DSP_SETPARAM_FLOAT_CALLBACK setparameterfloat;
  FMOD_DSP_SETPARAM_INT_CALLBACK setparameterint;
  FMOD_DSP_SETPARAM_BOOL_CALLBACK setparameterbool;
  FMOD_DSP_SETPARAM_DATA_CALLBACK setparameterdata;
  FMOD_DSP_GETPARAM_FLOAT_CALLBACK getparameterfloat;
  FMOD_DSP_GETPARAM_INT_CALLBACK getparameterint;
  FMOD_DSP_GETPARAM_BOOL_CALLBACK getparameterbool;
  FMOD_DSP_GETPARAM_DATA_CALLBACK getparameterdata;
  FMOD_DSP_SHOULDIPROCESS_CALLBACK shouldiprocess;
  void *userdata;
  FMOD_DSP_SYSTEM_REGISTER_CALLBACK sys_register;
  FMOD_DSP_SYSTEM_DEREGISTER_CALLBACK sys_deregister;
  FMOD_DSP_SYSTEM_MIX_CALLBACK sys_mix;
} FMOD_DSP_DESCRIPTION;

JavaScript Syntax

struct FMOD_DSP_DESCRIPTION {
  pluginsdkversion,
  name,
version,
numinputbuffers,
numoutputbuffers,
create,
release,
reset,
read,
process,
setposition,
numparameters,
setparameterfloat,
setparameterint,
setparameterbool,
setparameterdata,
getparameterfloat,
getparameterint,
getparameterbool,
getparameterdata,
shouldiprocess,
userdata,
sys_register,
sys_deregister,
sys_mix,
Members

pluginsdkversion

[w] The plugin SDK version this plugin is built for. Set to this to
FMOD_PLUGIN_SDK_VERSION defined above.

name

[w] The identifier of the DSP. This will also be used as the name of DSP and
shouldn't change between versions.

version

[w] Plugin writer's version number.

numinputbuffers

[w] Number of input buffers to process. Use 0 for DSPs that only generate sound
and 1 for effects that process incoming sound.

numoutputbuffers

[w] Number of audio output buffers. Only one output buffer is currently supported.

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.

process

[w] Process callback. Can be specified instead of the read callback if any channel format changes occur between input and output. This also replaces shouldProcess and should return an error if the effect is to be bypassed. 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.

setParameterFloat

[w] This is called when the user calls DSP::setParameterFloat. Can be null.

setParameterInt

[w] This is called when the user calls DSP::setParameterInt. Can be null.

setParameterBool

[w] This is called when the user calls DSP::setParameterBool. Can be null.

setParameterData
This is called when the user calls `DSP::setParameterData`. Can be null.

`getparameterfloat`

This is called when the user calls `DSP::getParameterFloat`. Can be null.

`getparameterint`

This is called when the user calls `DSP::getParameterInt`. Can be null.

`getparameterbool`

This is called when the user calls `DSP::getParameterBool`. Can be null.

`getparameterdata`

This is called when the user calls `DSP::getParameterData`. Can be null.

`shouldprocess`

This is called before processing. You can detect if inputs are idle and return `FMOD_OK` to process, or any other error code to avoid processing the effect. Use a count down timer to allow effect tails to process before idling!

`userdata`

Optional. Specify 0 to ignore. This is user data to be attached to the DSP unit during creation. Access via `FMOD_DSP_STATE_FUNCTIONS::getuserdata`.

`sys_register`

Register callback. This is called when DSP unit is loaded/registered. Useful for 'global'/per system object init for plugin. Can be null.

`sys_deregister`

Deregister callback. This is called when DSP unit is unloaded/deregistered. Useful as 'global'/per system object shutdown for plugin. Can be null.

`sys_mix`
[w] System mix stage callback. This is called when the mixer starts to execute or is just finishing executing. Useful for 'global'/per system object once a mix update calls for a plugin. Can be null.
**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::setParameterFloat / DSP::setParameterInt / DSP::setParameterBool / DSP::setParameterData`. 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.

**JavaScript only :**

Not all fields are currently supported or may not work as expected at this time. To initialize a new instance in javascript use "FMOD.DSP_DESCRIPTION()", no 'new' keyword is required.
See Also

- System::createDSP
- DSP::setParameterFloat
- DSP::setParameterInt
- DSP::setParameterBool
- DSP::setParameterData
- FMOD_DSP_STATE
- FMOD_DSP_CREATE_CALLBACK
- FMOD_DSP_RELEASE_CALLBACK
- FMOD_DSP_RESET_CALLBACK
- FMOD_DSP_READ_CALLBACK
- FMOD_DSP_PROCESS_CALLBACK
- FMOD_DSP_SETPOSITION_CALLBACK
- FMOD_DSP_PARAMETER_DESC
- FMOD_DSP_SETPARAM_FLOAT_CALLBACK
- FMOD_DSP_SETPARAM_INT_CALLBACK
- FMOD_DSP_SETPARAM_BOOL_CALLBACK
- FMOD_DSP_SETPARAM_DATA_CALLBACK
- FMOD_DSP_GETPARAM_FLOAT_CALLBACK
- FMOD_DSP_GETPARAM_INT_CALLBACK
- FMOD_DSP_GETPARAM_BOOL_CALLBACK
- FMOD_DSP_GETPARAM_DATA_CALLBACK
- FMOD_DSP_SHOULDIPROCESS_CALLBACK
- FMOD_DSP_SYSTEM_REGISTER_CALLBACK
- FMOD_DSP_SYSTEM_DEREGISTER_CALLBACK
- FMOD_DSP_SYSTEM_MIX_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_DSP_METERING_INFO

DSP metering info used for retrieving metering info with `DSP::getMeteringInfo`

C/C++ Syntax

typedef struct {
    int numsamples;
    float peaklevel[32];
    float rmslevel[32];
    short numchannels;
} FMOD_DSP_METERING_INFO;

JavaScript Syntax

struct FMOD_DSP_METERING_INFO
{
    numsamples,
    peaklevel,
    rmslevel,
    numchannels,
};
Members

numsamples

[r] The number of samples considered for this metering info.

peaklevel

[r] The peak level per channel.

rmslevel

[r] The rms level per channel.

numchannels

[r] Number of channels.
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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_METERING_INFO()", no 'new' keyword is required.
See Also

- FMOD_SPEAKER
- DSP::getMeteringInfo
Firelight Technologies FMOD Studio API
**FMOD_DSP_PARAMETER_3DATTRIBUTES**

Structure for data parameters of type
**FMOD_DSP_PARAMETER_DATA_TYPE_3DATTRIBUTES**.

A parameter of this type is used in effects that respond to a 3D position.

**C/C++ Syntax**

```c
typedef struct {
    FMOD_3D_ATTRIBUTES relative;
    FMOD_3D_ATTRIBUTES absolute;
} FMOD_DSP_PARAMETER_3DATTRIBUTES;
```
Members

relative

[w] The position of the sound relative to the listener.

absolute

[w] The position of the sound in world coordinates.
Remarks

The FMOD::Studio::System will set this parameter automatically if an FMOD::Studio::EventInstance position changes, however if using the low level FMOD::System you must set this DSP parameter explicitly.

Attributes must use a coordinate system with the positive Y axis being up and the positive X axis being right. FMOD will convert passed in coordinates to left-handed for the plugin if the System was initialized with the FMOD_INIT_3D_RIGHTHANDED flag.

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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_PARAMETER_3DATTRIBUTES()", no 'new' keyword is required.
See Also

- FMOD_DSP_PARAMETER_DATA_TYPE
- FMOD_DSP_PARAMETER_DESC
Firelight Technologies FMOD Studio API
**FMOD_DSP_PARAMETER_3DATTRIBUTES_MULTI**

Structure for data parameters of type `FMOD_DSP_PARAMETER_DATA_TYPE_3DATTRIBUTES_MULTI`.

A parameter of this type is used in effects that respond to a 3D position and support multiple listeners.

**C/C++ Syntax**

```c
typedef struct {
    int numlisteners;
    FMOD_3D_ATTRIBUTES relative[FMOD_MAX_LISTENERS];
    float weight[FMOD_MAX_LISTENERS];
    FMOD_3D_ATTRIBUTES absolute;
} FMOD_DSP_PARAMETER_3DATTRIBUTES_MULTI;
```

**JavaScript Syntax**

```javascript
struct FMOD_DSP_PARAMETER_3DATTRIBUTES_MULTI
{
    numlisteners,
};
```
Members

numlisteners

[w] The number of listeners.

relative

[w] The position of the sound relative to the listeners.

weight

[w] The weighting of the listeners where 0 means listener has no contribution and 1 means full contribution.

absolute

[w] The position of the sound in world coordinates.
Remarks

The FMOD::Studio::System will set this parameter automatically if an FMOD::Studio::EventInstance position changes, however if using the low level FMOD::System you must set this DSP parameter explicitly.

Attributes must use a coordinate system with the positive Y axis being up and the positive X axis being right. FMOD will convert passed in coordinates to left-handed for the plugin if the System was initialized with the FMOD_INIT_3D_RIGHTHANDED flag.

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.

**JavaScript only :**

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_PARAMETER_3DATTRIBUTES_MULTI()", no 'new' keyword is required.
See Also

- `FMOD_DSP_PARAMETER_DATA_TYPE`
- `FMOD_DSP_PARAMETER_DESC`
Firelight Technologies FMOD Studio API
FMOD_DSP_PARAMETER_DESC

Base Structure for DSP parameter descriptions.

C/C++ Syntax

typedef struct {
    FMOD_DSP_PARAMETER_TYPE type;
    char name[16];
    char label[16];
    const char *description;
    FMOD_DSP_PARAMETER_DESC_FLOAT floatdesc;
    FMOD_DSP_PARAMETER_DESC_INT intdesc;
    FMOD_DSP_PARAMETER_DESC_BOOL booldesc;
    FMOD_DSP_PARAMETER_DESC_DATA datadesc;
} FMOD_DSP_PARAMETER_DESC;

JavaScript Syntax

struct FMOD_DSP_PARAMETER_DESC
{
    type,
    name,
    label,
    description,
};
Members

type

[w] Type of this 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 parameter.

floatdesc

[w] Struct containing information about the parameter in floating point format. Use when type is FMOD_DSP_PARAMETER_TYPE_FLOAT.

intdesc

[w] Struct containing information about the parameter in integer format. Use when type is FMOD_DSP_PARAMETER_TYPE_INT.

booldesc

[w] Struct containing information about the parameter in boolean format. Use when type is FMOD_DSP_PARAMETER_TYPE_BOOL.

datadesc

[w] Struct containing information about the parameter in data format. Use when type is FMOD_DSP_PARAMETER_TYPE_DATA.
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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_PARAMETER_DESC()", no 'new' keyword is required.
See Also

- System::createDSP
- DSP::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterInt
- DSP::getParameterInt
- DSP::setParameterBool
- DSP::getParameterBool
- DSP::setParameterData
- DSP::getParameterData
- FMOD_DSP_PARAMETER_DESC_FLOAT
- FMOD_DSP_PARAMETER_DESC_INT
- FMOD_DSP_PARAMETER_DESC_BOOL
- FMOD_DSP_PARAMETER_DESC_DATA
- FMOD_DSP_PARAMETER_DATA_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_PARAMETER_DESC_BOOL

Structure to define a boolean parameter for a DSP unit.

C/C++ Syntax

typedef struct {
    FMOD_BOOL defaultval;
    const char* const* valuenames;
} FMOD_DSP_PARAMETER_DESC_BOOL;

JavaScript Syntax

struct FMOD_DSP_PARAMETER_DESC_BOOL {
    defaultval,
};
Members

`defaultval`

[w] Default parameter value.

`valuenames`

[w] Names for false and true, respectively. There should be two strings. Optional.
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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_PARAMETER_DESC_BOOL()", no 'new' keyword is required.
See Also

- System::createDSP
- DSP::setParameterBool
- DSP::getParameterBool
- FMOD_DSP_PARAMETER_DESC
Firelight Technologies FMOD Studio API
**FMOD_DSP_PARAMETER_DESC_DATA**

Structure to define a data parameter for a DSP unit. Use 0 or above for custom types. This parameter will be treated specially by the system if set to one of the `FMOD_DSP_PARAMETER_DATA_TYPE` values.

**C/C++ Syntax**

```c
typedef struct {
    int datatype;
} FMOD_DSP_PARAMETER_DESC_DATA;
```

**JavaScript Syntax**

```javascript
struct FMOD_DSP_PARAMETER_DESC_DATA
{
    datatype,
};
```
Members

datatype

[w] The type of data for this parameter. Use 0 or above for custom types or set to one of the FMOD_DSP_PARAMETER_DATA_TYPE values.
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.

JavaScript only :

Not all fields are currently supported or may not work as expected at this time.
To initialize an new instance in javascript use
"FMOD.DSP_PARAMETER_DESC_DATA()", no 'new' keyword is required.
See Also

- System::createDSP
- DSP::setParameterData
- DSP::getParameterData
- FMOD_DSP_PARAMETER_DATA_TYPE
- FMOD_DSP_PARAMETER_DESC
Firelight Technologies FMOD Studio API
**FMOD_DSP_PARAMETER_DESC_FLOAT**

Structure to define a float parameter for a DSP unit.

**C/C++ Syntax**

typedef struct {
    float min;
    float max;
    float defaultval;
    FMOD_DSP_PARAMETER_FLOAT_MAPPING mapping;
} FMOD_DSP_PARAMETER_DESC_FLOAT;

**JavaScript Syntax**

struct FMOD_DSP_PARAMETER_DESC_FLOAT
{
    min,
    max,
    defaultval,
};
Members

min
[w] Minimum parameter value.

max
[w] Maximum parameter value.

defaultval
[w] Default parameter value.

mapping
[w] How the values are distributed across dials and automation curves (e.g. linearly, exponentially etc).
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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize a new instance in javascript use "FMOD.DSP_PARAMETER_DESC_FLOAT()", no 'new' keyword is required.
See Also

- System::createDSP
- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_PARAMETER_DESC
- FMOD_DSP_PARAMETER_FLOAT_MAPPING
Firelight Technologies FMOD Studio API
**FMOD_DSP_PARAMETER_DESC_INT**

Structure to define a int parameter for a DSP unit.

**C/C++ Syntax**

```c
typedef struct {
    int min;
    int max;
    int defaultval;
    FMOD_BOOL goestoinf;
    const char* const* valuenames;
} FMOD_DSP_PARAMETER_DESC_INT;
```

**JavaScript Syntax**

```javascript
struct FMOD_DSP_PARAMETER_DESC_INT {
    min,
    max,
    defaultval,
    goestoinf,
};
```
**Members**

*min*

[w] Minimum parameter value.

*max*

[w] Maximum parameter value.

*defaultval*

[w] Default parameter value.

*goestoinf*

[w] Whether the last value represents infinity.

*valuenames*

[w] Names for each value. There should be as many strings as there are possible values (max - min + 1). Optional.
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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_PARAMETER_DESC_INT()", no 'new' keyword is required.
See Also

- `System::createDSP`
- `DSP::setParameterInt`
- `DSP::getParameterInt`
- `FMOD_DSP_PARAMETER_DESC`
Firelight Technologies FMOD Studio API
FMOD_DSP_PARAMETER_FFT

Structure for data parameters of type 
FMOD_DSP_PARAMETER_DATA_TYPE_FFT. A parameter of this type is 
declared for the FMOD_DSP_TYPE_FFT effect.

C/C++ Syntax

typedef struct {
    int length;
    int numchannels;
    float *spectrum[32];
} FMOD_DSP_PARAMETER_FFT;

JavaScript Syntax

struct FMOD_DSP_PARAMETER_FFT
{
    length,
    numchannels,
};
Members

length
[r] Number of entries in this spectrum window. Divide this by the output rate to get the hz per entry.

numchannels
[r] Number of channels in spectrum.

spectrum
[r] Per channel spectrum arrays. See remarks for more.
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.

Notes on the spectrum data member. Values inside the float buffer are typically between 0 and 1.0.
Each top level array represents one PCM channel of data.
Address data as spectrum[channel][bin]. A bin is 1 fft window entry.
Only read/display half of the buffer typically for analysis as the 2nd half is usually the same data reversed due to the nature of the way FFT works.

JavaScript only :

Not all fields are currently supported or may not work as expected at this time.
To initialize an new instance in javascript use
"FMOD.DSP_PARAMETER_FFT()", no 'new' keyword is required.
See Also

- FMOD_DSP_PARAMETER_DATA_TYPE
- FMOD_DSP_PARAMETER_DESC
- FMOD_DSP_PARAMETER_DATA_TYPE_FFT
- FMOD_DSP_TYPE
- FMOD_DSP_FFT
Firelight Technologies FMOD Studio API
FMOD_DSP_PARAMETER_FLOAT

Structure to define a mapping for a DSP unit's float parameter.

C/C++ Syntax

typedef struct {
    FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE type;
    FMOD_DSP_PARAMETER_FLOAT_MAPPING_PIECEWISE_LINEAR piecewiselinearmapping
} FMOD_DSP_PARAMETER_FLOAT_MAPPING;

JavaScript Syntax

struct FMOD_DSP_PARAMETER_FLOAT_MAPPING
{
    type,
};
Members

type

piecewiselinearmapping

[w] Only required for FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE_PIECEWISE_LINEAR
type mapping.
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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_PARAMETER_FLOAT_MAPPING()", no 'new' keyword is required.
See Also

- FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE
- FMOD_DSP_PARAMETER_FLOAT_MAPPING PIECEWISE_LINEAR
- FMOD_DSP_PARAMETER_DESC_FLOAT
Firelight Technologies FMOD Studio API
**FMOD_DSP_PARAMETER_FLOAT_MAPPING_PIECEWISE_LINEAR**

Structure to define a piecewise linear mapping.

**C/C++ Syntax**

```c
typedef struct {
    int numpoints;
    float *pointparamvalues;
    float *pointpositions;
} FMOD_DSP_PARAMETER_FLOAT_MAPPING_PIECEWISE_LINEAR;
```

**JavaScript Syntax**

```javascript
struct FMOD_DSP_PARAMETER_FLOAT_MAPPING_PIECEWISE_LINEAR {
    numpoints,
};
```
Members

\textit{numpoints}

[w] The number of pairs in the piecewise mapping (at least 2).

\textit{pointparamvalues}

[w] The values in the parameter's units for each point

\textit{pointpositions}

[w] The positions along the control's scale (e.g. dial angle) corresponding to each parameter value. The range of this scale is arbitrary and all positions will be relative to the minimum and maximum values (e.g. \([0,1,3]\) is equivalent to \([1,2,4]\) and \([2,4,8]\)). If this array is zero, pointparamvalues will be distributed with equal spacing.
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.

**JavaScript only:**

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_PARAMETER_FLOAT_MAPPING_PIECEWISE_LINEAR()", no 'new' keyword is required.
See Also

- **FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE**
- **FMOD_DSP_PARAMETER_FLOAT_MAPPING**
Firelight Technologies FMOD Studio API
FMOD_DSP_PARAMETER_OVERALL_GAIN

Structure for data parameters of type
FMOD_DSP_PARAMETER_DATA_TYPE_OVERALL_GAIN. A parameter of
this type is used in effects that affect the overgain of the signal in a predictable
way. This parameter is read by the system to determine the effect's gain for voice
virtualization.

C/C++ Syntax

typedef struct {
    float linear_gain;
    float linear_gain_additive;
} FMOD_DSP_PARAMETER_OVERALL_GAIN;

JavaScript Syntax

struct FMOD_DSP_PARAMETER_OVERALL_GAIN
{
    linear_gain,
    linear_gain_additive,
};
Members

*linear_gain*

[r] The overall linear gain of the effect on the direct signal path

*linear_gain_additive*

[r] Additive gain, for parallel signal paths
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.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize a new instance in javascript use "FMOD.DSP_PARAMETER_OVERALLGAIN()", no 'new' keyword is required.
See Also

- FMOD_DSP_PARAMETER_DATA_TYPE
- FMOD_DSP_PARAMETER_DESC
Firelight Technologies FMOD Studio API
FMOD_DSP_PARAMETER_SIDECHAIN

Structure for data parameters of type FMOD_DSP_PARAMETER_DATA_TYPE_SIDECHAIN. A parameter of this type is declared for effects which support sidechaining.

C/C++ Syntax

typedef struct {
    FMOD_BOOL sidechainenable;
} FMOD_DSP_PARAMETER_SIDECHAIN;

JavaScript Syntax

struct FMOD_DSP_PARAMETER_SIDECHAIN {
    sidechainenable,
};
Members

sidechainenable

[r/w] Whether sidechains are enabled.
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.

JavaScript only :

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_PARAMETER_SIDECHAIN()", no 'new' keyword is required.
See Also

- FMOD_DSP_PARAMETER_DATA_TYPE
- FMOD_DSP_PARAMETER_DESC
Firelight Technologies FMOD Studio API
FMOD_DSP_STATE

DSP plugin structure that is passed into each callback.

**C/C++ Syntax**

```c
typedef struct {
    void *instance;
    void *plugindata;
    FMOD_CHANNELMASK channelmask;
    FMOD_SPEAKERMODE source_speakermode;
    float *sidechainedata;
    int sidechainchannels;
    FMOD_DSP_STATE_FUNCTIONS *functions;
    int systemobject;
} FMOD_DSP_STATE;
```

**JavaScript Syntax**

```javascript
struct FMOD_DSP_STATE {
    plugindata,
    channelmask,
    source_speakermode,
    sidechainchannels,
    systemobject,
};
```
Members

instance

[r] Internal instance pointer, should not be used or written to.

plugindata

[w] Plugin writer created data the output author wants to attach to this object.

channelmask

[r] Specifies which speakers the DSP effect is active on

source_speakermode

[r] Specifies which speaker mode the signal originated for information purposes, ie in case panning needs to be done differently.

sidechaindata

[r] The mixed result of all incoming sidechains is stored at this pointer address.

sidechainchannels

[r] The number of channels of pcm data stored within the sidechain buffer.

functions

[r] Struct containing functions to give plugin developers the ability to query system state, access system level functionality and helpers.

systemobject

[r] FMOD::System object index, relating to the System object that created this DSP.
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.

'systemobject' is an integer that relates to the System object that created the DSP or registered the DSP plugin. If only 1 System object is created then it should be 0. A second object would be 1 and so on.

FMOD_DSP_STATE_FUNCTIONS::get_samplerate/get_blocksize/get_speakermode could return different results so it could be relevant to plugin developers to monitor which object is being used.

JavaScript only :

Not all fields are currently supported or may not work as expected at this time. To initialize a new instance in javascript use "FMOD.DSP_STATE()", no 'new' keyword is required.
See Also

- [FMOD_DSP_DESCRIPTION](#)
- [FMOD_DSP_STATE_FUNCTIONS](#)
Firelight Technologies FMOD Studio API
**FMOD_DSP_STATE_DFT_FUNCTIONS**

Struct containing DFT functions to enable a plugin to perform optimized time-frequency domain conversion.

**C/C++ Syntax**

```c
typedef struct {
    FMOD_DSP_DFT_FFTREAL_FUNC fftreal;
    FMOD_DSP_DFT_IFFTREAL_FUNC inversefftreal;
} FMOD_DSP_STATE_DFT_FUNCTIONS;
```
Members

fftreal

[r] Function for performing an FFT on a real signal.

inversefftreal

[r] Function for performing an inverse FFT to get a real signal.
Remarks

Members marked with [r] mean read only for the developer, read/write for the FMOD system.

Members marked with [w] mean read/write for the developer, read only for the FMOD system.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_STATE_DFT_FUNCTIONS()", no 'new' keyword is required.
See Also

- FMOD_DSP_STATE_FUNCTIONS
Firelight Technologies FMOD Studio API
FMOD_DSP_STATE_FUNCTIONS

Struct containing functions to give plugin developers the ability to query system state, access system level functionality and helpers.

C/C++ Syntax

typedef struct {
    FMOD_DSP_ALLOC_FUNC alloc;
    FMOD_DSP_REALLOC_FUNC realloc;
    FMOD_DSP_FREE_FUNC free;
    FMOD_DSP_GETSAMPLERATE_FUNC getsamplerate;
    FMOD_DSP_GETBLOCKSIZE_FUNC getblocksize;
    FMOD_DSP_STATE_DFT_FUNCTIONS *dft;
    FMOD_DSP_STATE_PAN_FUNCTIONS *pan;
    FMOD_DSP_GETSPEAKERMODE_FUNC getspeakermode;
    FMOD_DSP_GETCLOCK_FUNC getclock;
    FMOD_DSP_GETLISTENERATTRIBUTES_FUNC getlistenerattributes;
    FMOD_DSP_LOG_FUNC log;
    FMOD_DSP_GETUSERDATA_FUNC getuserdata;
} FMOD_DSP_STATE_FUNCTIONS;
Members

*alloc*

[r] Function to allocate memory using the FMOD memory system.

*realloc*

[r] Function to reallocate memory using the FMOD memory system.

*free*

[r] Function to free memory allocated with FMOD_DSP_ALLOC_FUNC.

*getsamplerate*

[r] Function to query the system sample rate.

*getblocksize*

[r] Function to query the system block size, DSPs will be requested to process blocks of varying length up to this size.

*dft*

[r] Struct containing DFT functions to enable a plugin to perform optimized time-frequency domain conversion.

*pan*

[r] Struct containing panning helper functions for spatialization plugins.

*getspeakermode*

[r] Function to query the system speaker modes. One is the mixer's default speaker mode, the other is the output mode the system is downmixing or upmixing to.

*getclock*
[r] Function to get the clock of the current DSP, as well as the subset of the input buffer that contains the signal.

getlistenerattributes

[r] Callback for getting the absolute listener attributes set via the API (returned as left-handed coordinates).

log

[r] Function to write to the FMOD logging system.

getuserdata

[r] Function to get the user data attached to this DSP. See FMOD_DSP_DESCRIPTION::userdata.
Remarks

Members marked with [r] mean read only for the developer, read/write for the FMOD system.

Members marked with [w] mean read/write for the developer, read only for the FMOD system.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_STATE_FUNCTIONS()", no 'new' keyword is required.
See Also

- FMODE_DSP_STATE
- FMOD_DSP_STATE_DFT_FUNCTIONS
- FMOD_DSP_STATE_PAN_FUNCTIONS
Firelight Technologies FMOD Studio API
**FMOD_DSP_STATE_PAN_FUNCTIONS**

Struct containing panning helper functions for spatialization plugins.

**C/C++ Syntax**

```c
typedef struct {
    FMOD_DSP_PAN_SUMMONOMATRIX_FUNC summonomatrix;
    FMOD_DSP_PAN_SUMSTEREOMATRIX_FUNC sumstereomatrix;
    FMOD_DSP_PAN_SUMSURROUNDMATRIX_FUNC sumsurroundmatrix;
    FMOD_DSP_PAN_SUMMONOTOSURROUNDMATRIX_FUNC summonotosurroundmatrix;
    FMOD_DSP_PAN_SUMSTEREOTOSURROUNDMATRIX_FUNC sumstereotosurroundmatrix;
    FMOD_DSP_PAN_GETROLLOFFGAIN_FUNC getrolloffgain;
} FMOD_DSP_STATE_PAN_FUNCTIONS;
```
Members

*summonomatrix*

[r] TBD.

*sumstereomatrix*

[r] TBD.

*sumsurroundmatrix*

[r] TBD.

*summonotosurroundmatrix*

[r] TBD.

*sumstereotosurroundmatrix*

[r] TBD.

*getrolloffgain*

[r] TBD.
Remarks

These are experimental, please contact support@fmod.org for more information.

Members marked with [r] mean read only for the developer, read/write for the FMOD system.

Members marked with [w] mean read/write for the developer, read only for the FMOD system.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.DSP_STATE_PAN_FUNCTIONS()", no 'new' keyword is required.
See Also

- FMOD_DSP_STATE_FUNCTIONS
- FMOD_DSP_PAN_SURROUND_FLAGS
Firelight Technologies FMOD Studio API
**FMOD_ERRORCALLBACK_INFO**

Structure that is passed into `FMOD_SYSTEM_CALLBACK` for the `FMOD_SYSTEM_CALLBACK_ERROR` callback type.

**C/C++ Syntax**

```c
typedef struct {
    FMOD_RESULT result;
    FMOD_ERRORCALLBACK_INSTANCETYPE instancetype;
    void *instance;
    const char *functionname;
    const char *functionparams;
} FMOD_ERRORCALLBACK_INFO;
```

**JavaScript Syntax**

```javascript
struct FMOD_ERRORCALLBACK_INFO {
    result,
    instancetype,
    instance,
    functionname,
    functionparams,
};
```
**Members**

*result*

Error code result

*instancetype*

Type of instance the error occurred on

*instance*

Instance pointer

*functionname*

Function that the error occurred on

*functionparams*

Function parameters that the error occurred on
Remarks

The instance pointer will be a type corresponding to the instanceType enum.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.ERRORCALLBACK_INFO()", no 'new' keyword is required.
See Also

- `FMOD_ERRORCALLBACK_INSTANCETYPE`
Firelight Technologies FMOD Studio API
**FMOD_GUID**

Structure describing a globally unique identifier.

**C/C++ Syntax**

```c
typedef struct {
    unsigned int Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
} FMOD_GUID;
```

**JavaScript Syntax**

```javascript
struct FMOD_GUID {
    Data1,
    Data2,
    Data3,
    Data4,
};
```
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.
Remarks

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.GUID()", no 'new' keyword is required.
See Also

- System::getDriverInfo
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_DESCRIPTION

When creating an output, declare one of these and provide the relevant callbacks and name for FMOD to use when it creates and uses an output of this type.

C/C++ Syntax

typedef struct {
    unsigned int apiversion;
    const char *name;
    unsigned int version;
    int polling;
    FMOD_OUTPUT_GETNUMDRIVERS_CALLBACK getnumdrivers;
    FMOD_OUTPUT_GETDRIVERINFO_CALLBACK getdriverinfo;
    FMOD_OUTPUT_INIT_CALLBACK init;
    FMOD_OUTPUT_START_CALLBACK start;
    FMOD_OUTPUT_STOP_CALLBACK stop;
    FMOD_OUTPUT_CLOSE_CALLBACK close;
    FMOD_OUTPUT_UPDATE_CALLBACK update;
    FMOD_OUTPUT_GETHANDLE_CALLBACK gethandle;
    FMOD_OUTPUT_GETPOSITION_CALLBACK getposition;
    FMOD_OUTPUT_LOCK_CALLBACK lock;
    FMOD_OUTPUT_UNLOCK_CALLBACK unlock;
    FMOD_OUTPUT_MIXER_CALLBACK mixer;
    FMOD_OUTPUT_OBJECT3DGETINFO_CALLBACK object3dgetinfo;
    FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK object3dalloc;
    FMOD_OUTPUT_OBJECT3DFREE_CALLBACK object3dfree;
    FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK object3dupdate;
    FMOD_OUTPUT_OPENPORT_CALLBACK openport;
    FMOD_OUTPUT_CLOSEPORT_CALLBACK closeport;
} FMOD_OUTPUT_DESCRIPTION;

JavaScript Syntax

struct FMOD_OUTPUT_DESCRIPTION
{
    apiversion,
    name,
    version,
    polling,
    getnumdrivers,
    getdriverinfo,
    init,
start,
stop,
close,
update,
gethandle,
getposition,
lock,
unlock,
mixer,
object3dgetinfo,
object3dalloc,
object3dfree,
object3dupdate,
openport,
closeport,

};
Members

`apiversion`

[w] The output plugin API version this plugin is built for. Set to this to `FMOD_OUTPUT_PLUGIN_VERSION`.

`name`

[w] Name of the output plugin.

`version`

[w] Version of the output plugin.

`polling`

[w] If TRUE (non-zero) a mixer thread is created that calls `FMOD_OUTPUT_GETPOSITION_CALLBACK` / `FMOD_OUTPUT_LOCK_CALLBACK` / `FMOD_OUTPUT_UNLOCK_CALLBACK` to drive the mixer. If FALSE (zero) you must call `FMOD_OUTPUT_READFROMMIXER` to drive the mixer yourself.

`getnumdrivers`

[w] Required user thread callback to provide the number of attached sound devices. Called from `System::getNumDrivers`.

`getdriverinfo`

[w] Required user thread callback to provide information about a particular sound device. Called from `System::getDriverInfo`.

`init`

[w] Required user thread callback to allocate resources and provide information about hardware capabilities. Called from `System::init`. 
start

[w] Optional user thread callback just before mixing should begin, calls to
FMOD_OUTPUT_GETPOSITION_CALLBACK / 
FMOD_OUTPUT_LOCK_CALLBACK / 
FMOD_OUTPUT_UNLOCK_CALLBACK / 
FMOD_OUTPUT_MIXER_CALLBACK will start, you may call 
FMOD_OUTPUT_READFROMMIXER after this point. Called from
 System::init.

stop

[w] Optional user thread callback just after mixing has finished, calls to
FMOD_OUTPUT_GETPOSITION_CALLBACK / 
FMOD_OUTPUT_LOCK_CALLBACK / 
FMOD_OUTPUT_UNLOCK_CALLBACK / 
FMOD_OUTPUT_MIXER_CALLBACK have stopped, you may not call 
FMOD_OUTPUT_READFROMMIXER after this point. Called from
 System::close.

close

[w] Required user thread callback to clean up resources allocated during
FMOD_OUTPUT_INIT_CALLBACK. Called from System::init and
System::close.

update

[w] Optional user thread callback once per frame to update internal state. Called
from System::update.

gethandle

[w] Optional user thread callback to provide a pointer to the internal device
object used to share with other audio systems. Called from
System::getOutputHandle.

close

[w] Required mixer thread callback (if 'polling' is TRUE) to provide the
hardware playback position in the output ring buffer. Called before a mix.

lock

[w] Required mixer thread callback (if 'polling' is TRUE) to provide a pointer the mixer can write to for the next block of audio data. Called before a mix.

unlock

[w] Optional mixer thread callback (if 'polling' is TRUE) to signify the mixer has finished writing to the pointer from FMOD_OUTPUT_LOCK_CALLBACK. Called after a mix.

mixer

[w] Optional mixer thread callback (if 'polling' is FALSE) called repeatedly to give a thread for waiting on an audio hardware synchronization primitive (see remarks for details). Ensure you have a reasonable timeout (~200ms) on your synchronization primitive and allow this callback to return once per wakeup to avoid deadlocks.

object3dgetinfo

[w] Optional mixer thread callback to provide information about the capabilities of 3D object hardware. Called during a mix.

object3dalloc

[w] Optional mixer thread callback to reserve a hardware resources for a single 3D object. Called during a mix.

object3dfree

[w] Optional mixer thread callback to release a hardware resource previously acquired with FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK. Called during a mix.

object3dupdate

[w] Optional mixer thread callback once for every acquired 3D object every mix
to provide 3D information and buffered audio. Called during a mix.

openport

[w] Optional main thread callback to open an auxiliary output port on the device.

closeport

[w] Optional main thread callback to close an auxiliary output port on the device.
Remarks

There are several methods for driving the FMOD mixer to service the audio hardware.

- Polled: if the audio hardware must be polled regularly set 'polling' to TRUE, FMOD will create a mixer thread that calls back via \texttt{FMOD\_OUTPUT\_GETPOSITION\_CALLBACK}. Once an entire block of samples have played FMOD will call \texttt{FMOD\_OUTPUT\_LOCK\_CALLBACK} to allow you to provide a destination pointer to write the next mix.
- Callback: if the audio hardware provides a callback where you must provide a buffer of samples then set 'polling' to FALSE and directly call \texttt{FMOD\_OUTPUT\_READFROMMIXER}.
- Synchronization: if the audio hardware provides a synchronization primitive to wait on then set 'polling' to FALSE and give a \texttt{FMOD\_OUTPUT\_MIXER\_CALLBACK} pointer. FMOD will create a mixer thread and call you repeatedly once \texttt{FMOD\_OUTPUT\_START\_CALLBACK} has finished, you must wait on your primitive in this callback and upon wake call \texttt{FMOD\_OUTPUT\_READFROMMIXER}.
- Non-realtime: if you are writing a file or driving a non-realtime output call \texttt{FMOD\_OUTPUT\_READFROMMIXER} from \texttt{FMOD\_OUTPUT\_UPDATE\_CALLBACK}.

Callbacks marked with 'user thread' will be called in response to the user of the FMOD low level API, in the case of the Studio runtime API, the user is the Studio Update thread.

Members marked with [r] mean read only for the developer, read/write for the FMOD system.

Members marked with [w] mean read/write for the developer, read only for the FMOD system.

\textbf{JavaScript only :}

Not all fields are currently supported or may not work as expected at this time.
To initialize an new instance in javascript use
"FMOD.OUTPUT_DESCRIPTION()", no 'new' keyword is required.
See Also

- FMOD_OUTPUT_STATE
- FMOD_OUTPUT_GETNUMDRIVERS_CALLBACK
- FMOD_OUTPUT_GETDRIVERINFO_CALLBACK
- FMOD_OUTPUT_INIT_CALLBACK
- FMOD_OUTPUT_START_CALLBACK
- FMOD_OUTPUT_STOP_CALLBACK
- FMOD_OUTPUT_CLOSE_CALLBACK
- FMOD_OUTPUT_UPDATE_CALLBACK
- FMOD_OUTPUT_GETHANDLE_CALLBACK
- FMOD_OUTPUT_GETPOSITION_CALLBACK
- FMOD_OUTPUT_LOCK_CALLBACK
- FMOD_OUTPUT_UNLOCK_CALLBACK
- FMOD_OUTPUT_MIXER_CALLBACK
- FMOD_OUTPUT_OBJECT3DGETINFO_CALLBACK
- FMOD_OUTPUT_OBJECT3DALLOC_CALLBACK
- FMOD_OUTPUT_OBJECT3DFREE_CALLBACK
- FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_OBJECT3DINFO

This structure is passed to the plugin via FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK, so that whatever object based panning solution available can position it in the speakers correctly. Object based panning is a 3D panning solution that sends a mono only signal to a hardware device, such as Dolby Atmos or other similar panning solutions.

C/C++ Syntax

typedef struct {
  float *buffer;
  unsigned int bufferlength;
  FMOD_VECTOR position;
  float gain;
  float spread;
  float priority;
} FMOD_OUTPUT_OBJECT3DINFO;

JavaScript Syntax

struct FMOD_OUTPUT_OBJECT3DINFO
{
  bufferlength,
  gain,
  spread,
  priority,
};
Members

buffer
[r] Mono PCM floating point buffer. This buffer needs to be scaled by the gain value to get distance attenuation.

bufferlength
[r] Length in PCM samples of buffer.

position
[r] Vector relative between object and listener.

gain
[r] 0.0 to 1.0 - 1 = 'buffer' is not attenuated, 0 = 'buffer' is fully attenuated.

spread
[r] 0 - 360 degrees. 0 = point source, 360 = sound is spread around all speakers

priority
[r] 0.0 to 1.0 - 0 = most important, 1 = least important. Based on height and distance (height is more important).
Remarks

FMOD does not attenuate the buffer, but provides a 'gain' parameter that the user must use to scale the buffer by. Rather than pre-attenuating the buffer, the plugin developer can access untouched data for other purposes, like reverb sending for example. The 'gain' parameter is based on the user's 3D custom rolloff model.

Members marked with [r] mean read only for the developer, read/write for the FMOD system. Members marked with [w] mean read/write for the developer, read only for the FMOD system.

JavaScript only :

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use
"FMOD.OUTPUT_OBJECT3DINFO()", no 'new' keyword is required.
See Also

- `FMOD_OUTPUT_OBJECT3DUPDATE_CALLBACK`
Firelight Technologies FMOD Studio API
FMOD_OUTPUT_STATE

Output object state passed into every callback provides access to plugin developers data and system functionality.

C/C++ Syntax

typedef struct {
    void *plugindata;
    FMOD_OUTPUT_READFROMMIXER_FUNC readfrommixer;
    FMOD_OUTPUT_ALLOC_FUNC alloc;
    FMOD_OUTPUT_FREE_FUNC free;
    FMOD_OUTPUT_LOG_FUNC log;
    FMOD_OUTPUT_COPYPORT_FUNC copyport;
    FMOD_OUTPUT_REQUESTRESET_FUNC requestreset;
} FMOD_OUTPUT_STATE;

JavaScript Syntax

struct FMOD_OUTPUT_STATE
{
    plugindata,
};
Members

plugindata

[w] Pointer used to store any plugin specific state so it's available in all callbacks.

readfrommixer

[r] Function to execute the mixer producing a buffer of audio. Used to control when the mix occurs manually as an alternative to FMOD_OUTPUT_DESCRIPTION::polling == TRUE.

alloc

[r] Function to allocate memory using the FMOD memory system.

free

[r] Function to free memory allocated with FMOD_OUTPUT_ALLOC.

log

[r] Function to write to the FMOD logging system.

copyport

[r] Function to copy the output from the mixer for the given auxiliary port.

requestreset

[r] Function to request the output plugin be shutdown then restarted during the next System::update.
Remarks

Members marked with [r] mean read only for the developer, read/write for the FMOD system. Members marked with [w] mean read/write for the developer, read only for the FMOD system.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.OUTPUT_STATE()", no 'new' keyword is required.
See Also

- FMOD_OUTPUT_DESCRIPTION
Firelight Technologies FMOD Studio API
**FMOD_PLUGINLIST**

Used to support lists of plugins within the one file.

**C/C++ Syntax**

```c
typedef struct {
    FMOD_PLUGINTYPE type;
    void* description;
} FMOD_PLUGINLIST;
```

**JavaScript Syntax**

```javascript
struct FMOD_PLUGINLIST {
    type,
    description,
};
```
Members

type
The plugin type

description
One of FMOD_DSP_DESCRIPTION, FMOD_OUTPUT_DESCRIPTION, FMOD_CODEC_DESCRIPTION
Remarks

The description field is either a pointer to `FMOD_DSP_DESCRIPTION`, `FMOD_OUTPUT_DESCRIPTION`, `FMOD_CODEC_DESCRIPTION`.

This structure is returned from a plugin as a pointer to a list where the last entry has `FMOD_PLUGINTYPE_MAX` and a null description pointer.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.PLUGINLIST()", no 'new' keyword is required.
See Also

- System::getNumNestedPlugins
- System::getNestedPlugin
Firelight Technologies FMOD Studio API
**FMOD_REVERB_PROPERTIES**

Structure defining a reverb environment.

**C/C++ Syntax**

```c
typedef struct {
    float DecayTime;
    float EarlyDelay;
    float LateDelay;
    float HFReference;
    float HFDecayRatio;
    float Diffusion;
    float Density;
    float LowShelfFrequency;
    float LowShelfGain;
    float HighCut;
    float EarlyLateMix;
    float WetLevel;
} FMOD_REVERB_PROPERTIES;
```

**JavaScript Syntax**

```javascript
struct FMOD_REVERB_PROPERTIES {
    DecayTime,
    EarlyDelay,
    LateDelay,
    HFReference,
    HFDecayRatio,
    Diffusion,
    Density,
    LowShelfFrequency,
    LowShelfGain,
    HighCut,
    EarlyLateMix,
    WetLevel,
};
```
Members

DecayTime

[r/w] 0.0 20000.0 1500.0 Reverberation decay time (ms)

EarlyDelay

[r/w] 0.0 300.0 7.0 Initial reflection delay time (ms)

LateDelay

[r/w] 0.0 100 11.0 Late reverberation delay time relative to initial reflection (ms)

HFReference

[r/w] 20.0 20000.0 5000 Reference high frequency (Hz)

HFDelayRatio

[r/w] 10.0 100.0 50.0 High-frequency to mid-frequency decay time ratio (%)

Diffusion

[r/w] 0.0 100.0 100.0 Value that controls the echo density in the late reverberation decay (%)

Density

[r/w] 0.0 100.0 100.0 Value that controls the modal density in the late reverberation decay (%)

LowShelfFrequency

[r/w] 20.0 1000.0 250.0 Reference low frequency (Hz)

LowShelfGain

[r/w] -36.0 12.0 0.0 Relative room effect level at low frequencies (dB)
*HighCut*

[r/w] 20.0 20000.0 20000.0 Relative room effect level at high frequencies (Hz)

*EarlyLateMix*

[r/w] 0.0 100.0 50.0 Early reflections level relative to room effect (%)

*WetLevel*

[r/w] -80.0 20.0 -6.0 Room effect level at mid frequencies (dB)
Remarks

Note the default reverb properties are the same as the
**FMOD_PRESET_GENERIC** preset.

All members are read/write [r/w], written to by FMOD when queried with
**System::getReverbProperties** and read by FMOD when set with
**System::setReverbProperties**.

**JavaScript only:**

Not all fields are currently supported or may not work as expected at this time.
To initialize an new instance in javascript use
"FMOD.REVERB_PROPERTIES()", no 'new' keyword is required.
See Also

- System::setReverbProperties
- System::getReverbProperties
- FMOD_REVERB_PRESETS
Firelight Technologies FMOD Studio API
**FMOD_TAG**

Structure describing a piece of tag data.

**C/C++ Syntax**

```c
typedef struct {
    FMOD_TAGTYPE type;
    FMOD_TAGDATATYPE datatype;
    char *name;
    void *data;
    unsigned int datalen;
    FMOD_BOOL updated;
} FMOD_TAG;
```

**JavaScript Syntax**

```javascript
struct FMOD_TAG
{
    type,
    datatype,
    data,
    datalen,
    updated,
};
```
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.

**JavaScript only :**

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.TAG()", no 'new' keyword is required.
See Also

- `Sound::getTag`
- `FMOD_TAGTYPE`
- `FMOD_TAGDATATYPE`
Firelight Technologies FMOD Studio API


**FMOD_VECTOR**

Structure describing a point in 3D space.

**C/C++ Syntax**

```c
typedef struct {
    float x;
    float y;
    float z;
} FMOD_VECTOR;
```

**JavaScript Syntax**

```javascript
struct FMOD_VECTOR {
    x,
    y,
    z,
};
```
Members

\( x \)

\( x \) coordinate in 3D space.

\( y \)

\( y \) coordinate in 3D space.

\( z \)

\( z \) coordinate in 3D space.
Remarks

FMOD uses a left handed coordinate system by default. To use a right handed coordinate system specify

`FMOD_INIT_3D_RIGHTHANDED` from `FMOD_INITFLAGS` in `System::init`.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.VECTOR()", no 'new' keyword is required.
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 Studio API
Defines

FMOD_CHANNELMASK FMOD_CODEC_WAVEFORMAT_VERSION
FMOD_DEBUG_FLAGS
FMOD_DRIVER_STATE
FMOD_DSP_GETPARAM_VALUESTR_LENGTH
FMOD_INITFLAGS
FMOD_MAX_CHANNEL_WIDTH
FMOD_MAX_LISTENERS
FMOD_MAX_SYSTEMS
FMOD_MEMORY_TYPE
FMOD_MODE
FMOD_PORT_INDEX
FMOD_REVERB_MAXINSTANCES
FMOD_REVERB_PRESETS
FMOD_SYSTEM_CALLBACK_TYPE
FMOD_TIMEUNIT
Firelight Technologies FMOD Studio API
FMOD_CHANNELMASK

These are bitfields to describe for a certain number of channels in a signal, which channels are being represented. For example, a signal could be 1 channel, but contain the LFE channel only.

FMOD_CHANNELMASK_BACK_CENTER is not represented as an output speaker in fmod - but it is encountered in input formats and is down or upmixed appropriately to the nearest speakers.

C/C++ Syntax

#define FMOD_CHANNELMASK_FRONT_LEFT 0x00000001
#define FMOD_CHANNELMASK_FRONT_RIGHT 0x00000002
#define FMOD_CHANNELMASK_FRONT_CENTER 0x00000004
#define FMOD_CHANNELMASK_LOW_FREQUENCY 0x00000008
#define FMOD_CHANNELMASK_SURROUND_LEFT 0x00000010
#define FMOD_CHANNELMASK_SURROUND_RIGHT 0x00000020
#define FMOD_CHANNELMASK_BACK_LEFT 0x00000040
#define FMOD_CHANNELMASK_BACK_RIGHT 0x00000080
#define FMOD_CHANNELMASK_BACK_CENTER 0x00000100
#define FMOD_CHANNELMASK_MONO (FMOD_CHANNELMASK_FRONT_LEFT)
#define FMOD_CHANNELMASK_STEREO (FMOD_CHANNELMASK_FRONT_LEFT | FMOD_CHANNELMASK_FRONT_RIGHT)
#define FMOD_CHANNELMASK_LRC (FMOD_CHANNELMASK_FRONT_LEFT | FMOD_CHANNELMASK_FRONT_RIGHT | FMOD_CHANNELMASK_FRONT_CENTER)
#define FMOD_CHANNELMASK_QUAD (FMOD_CHANNELMASK_FRONT_LEFT | FMOD_CHANNELMASK_FRONT_RIGHT | FMOD_CHANNELMASK_SURROUND_LEFT | FMOD_CHANNELMASK_SURROUND_RIGHT)
#define FMOD_CHANNELMASK_5POINT1 (FMOD_CHANNELMASK_FRONT_LEFT | FMOD_CHANNELMASK_FRONT_RIGHT | FMOD_CHANNELMASK_FRONT_CENTER | FMOD_CHANNELMASK_LOW_FREQUENCY | FMOD_CHANNELMASK_SURROUND_LEFT | FMOD_CHANNELMASK_SURROUND_RIGHT)
#define FMOD_CHANNELMASK_5POINT1_REARS (FMOD_CHANNELMASK_FRONT_LEFT | FMOD_CHANNELMASK_FRONT_RIGHT | FMOD_CHANNELMASK_FRONT_CENTER | FMOD_CHANNELMASK_LOW_FREQUENCY | FMOD_CHANNELMASK_BACK_LEFT | FMOD_CHANNELMASK_BACK_RIGHT)
#define FMOD_CHANNELMASK_7POINT0 (FMOD_CHANNELMASK_FRONT_LEFT | FMOD_CHANNELMASK_FRONT_RIGHT | FMOD_CHANNELMASK_FRONT_CENTER | FMOD_CHANNELMASK_LOW_FREQUENCY | FMOD_CHANNELMASK_SURROUND_LEFT | FMOD_CHANNELMASK_SURROUND_RIGHT | FMOD_CHANNELMASK_BACK_LEFT | FMOD_CHANNELMASK_BACK_RIGHT)
#define FMOD_CHANNELMASK_7POINT1 (FMOD_CHANNELMASK_FRONT_LEFT | FMOD_CHANNELMASK_FRONT_RIGHT | FMOD_CHANNELMASK_FRONT_CENTER | FMOD_CHANNELMASK_LOW_FREQUENCY | FMOD_CHANNELMASK_SURROUND_LEFT | FMOD_CHANNELMASK_SURROUND_RIGHT | FMOD_CHANNELMASK_BACK_LEFT | FMOD_CHANNELMASK_BACK_RIGHT | FMOD_CHANNELMASK_BACK_CENTER)

JavaScript Syntax

FMOD.CHANNELMASK_FRONT_LEFT
FMOD.CHANNELMASK_FRONT_RIGHT
FMOD.CHANNELMASK_FRONT_CENTER
FMOD.CHANNELMASK_LOW_FREQUENCY
FMOD.CHANNELMASK_SURROUND_LEFT
FMOD.CHANNELMASK_SURROUND_RIGHT
FMOD.CHANNELMASK_BACK_LEFT
FMOD.CHANNELMASK_BACK_RIGHT
FMOD.CHANNELMASK_BACK_CENTER
FMOD.CHANNELMASK_MONO
FMOD.CHANNELMASK_STEREO
FMOD.CHANNELMASK_LRC
FMOD.CHANNELMASK_QUAD
FMOD.CHANNELMASK_SURROUND
FMOD.CHANNELMASK_5POINT1
FMOD.CHANNELMASK_5POINT1_REARS
FMOD.CHANNELMASK_7POINT0
FMOD.CHANNELMASK_7POINT1
Values

FMOD_CHANNELMASK_FRONT_LEFT
FMOD_CHANNELMASK_FRONT_RIGHT
FMOD_CHANNELMASK_FRONT_CENTER
FMOD_CHANNELMASK_LOW_FREQUENCY
FMOD_CHANNELMASK_SURROUND_LEFT
FMOD_CHANNELMASK_SURROUND_RIGHT
FMOD_CHANNELMASK_BACK_LEFT
FMOD_CHANNELMASK_BACK_RIGHT
FMOD_CHANNELMASK_BACK_CENTER
FMOD_CHANNELMASK_MONO
FMOD_CHANNELMASK_STEREO
FMOD_CHANNELMASK_LRC
FMOD_CHANNELMASK_QUAD
FMOD_CHANNELMASK_SURROUND
FMOD_CHANNELMASK_5POINT1
FMOD_CHANNELMASK_5POINT1_REARS
FMOD_CHANNELMASK_7POINT0
FMOD_CHANNELMASK_7POINT1
See Also

- DSP::setChannelFormat
- DSP::getChannelFormat
- FMOD_SPEAKERMODE
Firelight Technologies FMOD Studio API
**FMOD_CODEC_WAVEFORMAT_VERSION**

Version number of **FMOD_CODEC_WAVEFORMAT** structure. Should be set into **FMOD_CODEC_STATE** in the **FMOD_CODEC_OPEN_CALLBACK**.

Use this for binary compatibility and for future expansion.

**C/C++ Syntax**

```c
#define FMOD_CODEC_WAVEFORMAT_VERSION 3
```

**JavaScript Syntax**

```javascript
FMOD_CODEC_WAVEFORMAT_VERSION
```
Values

`FMOD_CODEC_WAVEFORMAT_VERSION`
See Also

- FMOD_CODEC_STATE
- FMOD_CODEC_DESCRIPTION
- FMOD_CODEC_OPEN_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_DEBUG_FLAGS

Specify the requested information to be output when using the logging version of FMOD.

C/C++ Syntax

```c
#define FMOD_DEBUG_LEVEL_NONE 0x00000000
#define FMOD_DEBUG_LEVEL_ERROR 0x00000001
#define FMOD_DEBUG_LEVEL_WARNING 0x00000002
#define FMOD_DEBUG_LEVEL_LOG 0x00000004
#define FMOD_DEBUG_TYPE_MEMORY 0x00000100
#define FMOD_DEBUG_TYPE_FILE 0x00000200
#define FMOD_DEBUG_TYPE_CODEC 0x00000400
#define FMOD_DEBUG_TYPE_TRACE 0x00000800
#define FMOD_DEBUG_DISPLAY_TIMESTAMPS 0x00010000
#define FMOD_DEBUG_DISPLAY_LINENUMBERS 0x00020000
#define FMOD_DEBUG_DISPLAY_THREAD 0x00040000
```

JavaScript Syntax

```javascript
FMOD.DEBUG_LEVEL_NONE
FMOD.DEBUG_LEVEL_ERROR
FMOD.DEBUG_LEVEL_WARNING
FMOD.DEBUG_LEVEL_LOG
FMOD.DEBUG_TYPE_MEMORY
FMOD.DEBUG_TYPE_FILE
FMOD.DEBUG_TYPE_CODEC
FMOD.DEBUG_TYPE_TRACE
FMOD.DEBUG_DISPLAY_TIMESTAMPS
FMOD.DEBUG_DISPLAY_LINENUMBERS
FMOD.DEBUG_DISPLAY_THREAD
```
Values

\texttt{FMOD\_DEBUG\_LEVEL\_NONE}
Disable all messages

\texttt{FMOD\_DEBUG\_LEVEL\_ERROR}
Enable only error messages.

\texttt{FMOD\_DEBUG\_LEVEL\_WARNING}
Enable warning and error messages.

\texttt{FMOD\_DEBUG\_LEVEL\_LOG}
Enable informational, warning and error messages (default).

\texttt{FMOD\_DEBUG\_TYPE\_MEMORY}
Verbose logging for memory operations, only use this if you are debugging a memory related issue.

\texttt{FMOD\_DEBUG\_TYPE\_FILE}
Verbose logging for file access, only use this if you are debugging a file related issue.

\texttt{FMOD\_DEBUG\_TYPE\_CODEC}
Verbose logging for codec initialization, only use this if you are debugging a codec related issue.

\texttt{FMOD\_DEBUG\_TYPE\_TRACE}
Verbose logging for internal errors, use this for tracking the origin of error codes.

\texttt{FMOD\_DEBUG\_DISPLAY\_TIMESTAMPS}
Display the time stamp of the log message in milliseconds.

*FMOD_DEBUG_DISPLAY_LINENUMBERS*

Display the source code file and line number for where the message originated.

*FMOD_DEBUG_DISPLAY_THREAD*

Display the thread ID of the calling function that generated the message.
See Also

- FMOD_Debug_Initialize
Firelight Technologies FMOD Studio API
**FMOD_DRIVER_STATE**

Flags that provide additional information about a particular driver.

**C/C++ Syntax**

```c
#define FMOD_DRIVER_STATE_CONNECTED 0x00000001
#define FMOD_DRIVER_STATE_DEFAULT 0x00000002
```

**JavaScript Syntax**

```javascript
FMOD.DRIVER_STATE_CONNECTED
FMOD.DRIVER_STATE_DEFAULT
```
Values

*FMOD_DRIVER_STATE_CONNECTED*

Device is currently plugged in.

*FMOD_DRIVER_STATE_DEFAULT*

Device is the users preferred choice.
See Also

- System::getRecordDriverInfo
**FMOD_DSP_GETPARAM_VALUESTR_LENGTH**

Length in bytes of the buffer pointed to by the valuestr argument of FMOD_DSP_GETPARAM_XXXX_CALLBACK functions.

DSP plugins should not copy more than this number of bytes into the buffer or memory corruption will occur.

**C/C++ Syntax**

```c
#define FMOD_DSP_GETPARAM_VALUESTR_LENGTH 32
```

**JavaScript Syntax**

```javascript
FMOD.DSP_GETPARAM_VALUESTR_LENGTH
```
Values

FMOD_DSP_GETPARAM_VALUESTR_LENGTH
See Also

- FMOD_DSP_GETPARAM_FLOAT_CALLBACK
- FMOD_DSP_GETPARAM_INT_CALLBACK
- FMOD_DSP_GETPARAM_BOOL_CALLBACK
- FMOD_DSP_GETPARAM_DATA_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_INITFLAGS

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.

C/C++ Syntax

#define FMOD_INIT_NORMAL 0x00000000
#define FMOD_INIT_STREAM_FROM_UPDATE 0x00000001
#define FMOD_INIT_MIX_FROM_UPDATE 0x00000002
#define FMOD_INIT_3D_RIGHTHANDED 0x00000004
#define FMOD_INIT_CHANNEL_LOWPASS 0x00000100
#define FMOD_INIT_CHANNEL_DISTANCEFILTER 0x00000200
#define FMOD_INIT_PROFILE_ENABLE 0x00010000
#define FMOD_INIT_VOL0_BECOMES_VIRTUAL 0x00020000
#define FMOD_INIT_GEOMETRY_USECLOSEST 0x00040000
#define FMOD_INIT_PREFER_DOLBY_DOWNMIX 0x00080000
#define FMOD_INIT_THREAD_UNSAFE 0x00100000
#define FMOD_INIT_PROFILE_METER_ALL 0x00200000
#define FMOD_INIT_DISABLE_SRS_HIGHPASSFILTER 0x00400000

JavaScript Syntax

FMOD.INIT_NORMAL
FMOD.INIT_STREAM_FROM_UPDATE
FMOD.INIT_MIX_FROM_UPDATE
FMOD.INIT_3D_RIGHTHANDED
FMOD.INIT_CHANNEL_LOWPASS
FMOD.INIT_CHANNEL_DISTANCEFILTER
FMOD.INIT_PROFILE_ENABLE
FMOD.INIT_VOL0_BECOMES_VIRTUAL
FMOD.INIT_GEOMETRY_USECLOSEST
FMOD.INIT_PREFER_DOLBY_DOWNMIX
FMOD.INIT_THREAD_UNSAFE
FMOD.INIT_PROFILE_METER_ALL
**Values**

*FMOD_INIT_NORMAL*

Initialize normally

*FMOD_INIT_STREAM_FROM_UPDATE*

No stream thread is created internally. Streams are driven from `System::update`. Mainly used with non-realtime outputs.

*FMOD_INIT_MIX_FROM_UPDATE*

No mixer thread is created internally. Mixing is driven from `System::update`. Only applies to polling based output modes such as `FMOD_OUTPUTTYPE_NOSOUND`, `FMOD_OUTPUTTYPE_WAVWRITER`, `FMOD_OUTPUTTYPE_DSOUND`, `FMOD_OUTPUTTYPE_WINMM`, `FMOD_OUTPUTTYPE_XAUDIO`.

*FMOD_INIT_3D_RIGHTHANDED*

3D calculations will be performed in right-handed coordinates.

*FMOD_INIT_CHANNEL_LOWPASS*

All `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. This also causes sounds to sound duller when the sound goes behind the listener, as a fake HRTF style effect. Use `System::setAdvancedSettings` to disable or adjust cutoff frequency for this feature.

*FMOD_INIT_CHANNEL_DISTANCEFILTER*

All `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_PROFILE_ENABLE*
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_BECOMESVIRTUAL**

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_GEOMETRY_USECLOSEST**

With the geometry engine, only process the closest polygon rather than accumulating all polygons the sound to listener line intersects.

**FMOD_INIT_PREFER_DOLBY_DOWNMIX**

When using `FMOD_SPEAKERMODE_5POINT1` with a stereo output device, use the Dolby Pro Logic II downmix algorithm instead of the SRS Circle Surround algorithm.

**FMOD_INIT_THREAD_UNSAFE**

Disables thread safety for API calls. Only use this if FMOD low level is being called from a single thread, and if Studio API is not being used!

**FMOD_INIT_PROFILE_METER_ALL**

Slower, but adds level metering for every single DSP unit in the graph. Use `DSP::setMeteringEnabled` to turn meters off individually.

**FMOD_INIT_DISABLE_SRS_HIGHPASSFILTER**

Using `FMOD_SPEAKERMODE_5POINT1` with a stereo output device will enable the SRS Circle Surround downmixer. By default the SRS downmixer applies a high pass filter with a cutoff frequency of 80Hz. Use this flag to diable the high pass filter, or use `FMOD_INIT_PREFER_DOLBY_DOWNMIX` to use the Dolby Pro Logic II downmix algorithm instead.
See Also

- System::init
- System::update
- System::setAdvancedSettings
- Channel::set3DOcclusion
Firelight Technologies FMOD Studio API
FMOD_MAX_CHANNEL_WIDTH

The maximum number of channels per frame of audio supported by audio files, buffers, connections and DSPs.

C/C++ Syntax

#define FMOD_MAX_CHANNEL_WIDTH 32

JavaScript Syntax

FMOD.MAX_CHANNEL_WIDTH
Values

`FMOD_MAX_CHANNEL_WIDTH`
See Also

- `FMOD_CHANNELORDER`
- `FMOD_CREATESOUNDEXINFO`
- `System::setSoftwareFormat`
- `System::getDefaultMixMatrix`
- `ChannelControl::setMixMatrix`
- `ChannelControl::getMixMatrix`
- `FMOD::DSP::setChannelFormat`
Firelight Technologies FMOD Studio API
**FMOD_MAX_LISTENERS**

The maximum number of listeners supported.

**C/C++ Syntax**

```c
#define FMOD_MAX_LISTENERS 8
```

**JavaScript Syntax**

```javascript
FMOD.MAX_LISTENERS
```
Values

*FMODE_MAX_LISTENERS*
See Also

- System::set3DNumListeners
- System::set3DListenerAttributes
- System::get3DListenerAttributes
Firelight Technologies FMOD Studio API
**FMOD_MAX_SYSTEMS**

The maximum number of FMOD::System objects allowed.

**C/C++ Syntax**

```
#define FMOD_MAX_SYSTEMS 8
```
Values

`FMOD_MAX_SYSTEMS`
See Also

- System_Create
Firelight Technologies FMOD Studio API
**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.

**C/C++ Syntax**

```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_BUFFER 0x00000008
#define FMOD_MEMORY_PLUGIN 0x00000010
#define FMOD_MEMORY_XBOX360_PHYSICAL 0x00100000
#define FMOD_MEMORY_PERSISTENT 0x00200000
#define FMOD_MEMORY_SECONDARY 0x00400000
#define FMOD_MEMORY_ALL 0xFFFFFFFF
```

**JavaScript Syntax**

```javascript
FMOD.MEMORY_NORMAL
FMOD.MEMORY_STREAM_FILE
FMOD.MEMORY_STREAM_DECODE
FMOD.MEMORY_SAMPLEDATA
FMOD.MEMORY_DSP_BUFFER
FMOD.MEMORY_PLUGIN
FMOD.MEMORY_XBOX360_PHYSICAL
FMOD.MEMORY_PERSISTENT
FMOD.MEMORY_SECONDARY
FMOD.MEMORY_ALL
```
**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\_CREATESOUNDEXINFO::decodebuffersize`.

*FMOD\_MEMORY\_SAMPLEDATA*

Sample data buffer. Raw audio data, usually PCM/MPEG/ADPCM/XMA data.

*FMOD\_MEMORY\_DSP\_BUFFER*

DSP memory block allocated when more than 1 output exists on a DSP node.

*FMOD\_MEMORY\_PLUGIN*

Memory allocated by a third party plugin.

*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_ALLOC_CALLBACK
- FMOD_MEMORY_REALLOC_CALLBACK
- FMOD_MEMORY_FREE_CALLBACK
- Memory_Initialize
Firelight Technologies FMOD Studio API
**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_OPENRAW`) 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_OPENRAW` 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.

**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.

**C/C++ Syntax**
```c
#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_CREATESTREAM 0x00000080
#define FMOD_CREATESAMPLE 0x00000100
#define FMOD_CREATECOMPRESSEDSAMPLE 0x00000200
#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_INVERSETAPEREDROLLOFF 0x00800000
#define FMOD_3D_CUSTOMROLLOFF 0x04000000
#define FMOD_3D_IGNOREGEOMETRY 0x40000000
#define FMOD_IGNORETAGS 0x02000000
#define FMOD_LOWMEM 0x08000000
#define FMOD_LOADSECONDARYRAM 0x20000000
#define FMOD_VIRTUAL_PLAYFROMSTART 0x80000000
```

**JavaScript Syntax**

FMOD.DEFAULT
FMODLOOP_OFF
FMOD_LOOP_NORMAL
FMOD_LOOP_BIDI
FMOD_2D
FMOD_3D
FMOD_CREATESTREAM
FMOD_CREATESAMPLE
FMOD_CREATECOMPRESSEDSAMPLE
FMOD_OPENUSER
FMOD_OPENMEMORY
FMOD_OPENMEMORY_POINT
FMOD_OPENRAW
FMOD_OPENONLY
FMOD_ACCURATETIME
FMOD_MPEGSEARCH
FMOD.NONBLOCKING
FMOD.UNIQUE
FMOD._3D_HEADRELATIVE
FMOD._3D_WORLDRELATIVE
FMOD._3D_INVERSEROLLOFF
FMOD._3D_LINEARROLLOFF
FMOD._3D_LINEAR_SQUAREROLLOFF
FMOD._3D_INVERSE_TAPEREDROLLOFF
FMOD._3D_CUSTOMROLLOFF
FMOD._3D_IGNOREGEOMETRY
FMOD.IGNORETAGS
FMOD.LOWMEM
FMOD.LOADSECONDARYRAM
FMOD.VIRTUAL_PLAYFROMSTART
Values

**FMOD_DEFAULT**

Default for all modes listed below. **FMOD_LOOP_OFF, FMOD_2D, FMOD_3D_WORLDRELATIVE, FMOD_3D_INVERSEROLLOFF**

**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_CREATESTREAM**

Decompress at runtime, streaming from the source provided (ie from disk). Overrides **FMOD_CREATESAMPLE** and **FMOD_CREATECOMPRESSESAMPLE**. 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 playback and most flexible.

**FMOD_CREATECOMPRESSEDSAMPLE**

Load MP2/MP3/FADPCM/IMAADPCM/Vorbis/AT9 or XMA into memory and leave it compressed. Vorbis/AT9/FADPCM encoding only supported in the .FSB container format. During playback the FMOD software mixer will decode it in realtime as a 'compressed sample'. Overrides **FMOD_CREATERSAMPLE**. If the sound data is not one of the supported formats, it will behave as if it was created with **FMOD_CREATERSAMPLE** and decode the sound into PCM.

**FMOD_OPENUSER**

Opens a user created static sample or stream. Use **FMOD_CREATERSOUNDEXINFO** 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_CREATERSOUNDEXINFO** to specify length. If used with **FMOD_CREATERSAMPLE** or **FMOD_CREATECOMPRESSEDSAMPLE**, 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_CREATERSOUNDEXINFO** to specify length. This differs to **FMOD_OPENMEMORY** in that it uses the memory as is, without duplicating the memory into its own buffers. Cannot be freed after open, only after **Sound::release**. Will not work if the data is compressed and **FMOD_CREATECOMPRESSEDSAMPLE** is not used.

**FMOD_OPENRAW**
Will ignore file format and treat as raw pcm. Use **FMOD_CREATE_SOUNDEXINFO** to specify format. Requires at least default frequency, num channels 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_ACCURATE_TIME**

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_INVERSE_ROLLOFF**
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.

**FMOD\_3D\_LINEARSQUAREROLLOFF**

This sound will follow a linear-square rolloff model where mindistance = full volume, maxdistance = silence.

**FMOD\_3D\_INVERSETAPEREDROLLOFF**

This sound will follow the inverse rolloff model at distances close to mindistance and a linear-square rolloff close to maxdistance.

**FMOD\_3D\_CUSTOMROLLOFF**

This sound will follow a rolloff model defined by [Sound::set3DCustomRolloff](https://www.fmod.com/doc/3.1/api/FMOD\_Sound.html#set3DCustomRolloff) / [Channel::set3DCustomRolloff](https://www.fmod.com/doc/3.1/api/FMOD\_Channel.html#set3DCustomRolloff).

**FMOD\_3D\_IGNOREGEOMETRY**

Is not affect by geometry occlusion. If not specified in [Sound::setMode](https://www.fmod.com/doc/3.1/api/FMOD\_Sound.html#setMode), or [Channel::setMode](https://www.fmod.com/doc/3.1/api/FMOD\_Channel.html#setMode), the flag is cleared and it is affected by geometry again.

**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](https://www.fmod.com/doc/3.1/api/FMOD\_Sound.html#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`
Firelight Technologies FMOD Studio API
FMOD_PORT_INDEX

C/C++ Syntax

#define FMOD_PORT_INDEX_NONE 0xFFFFFFFFFFFFFFFF

JavaScript Syntax

FMOD.PORT_INDEX_NONE
Values

_FMOD_PORT_INDEX_NONE_

Use when a port index is not required
See Also

- System::AttachChannelGroupToPort
Firelight Technologies FMOD Studio API
FMOD_REVERB_MAXINSTANCES

The maximum number of global/physical reverb instances.

Each instance of a physical reverb is an instance of a 
FMOD_DSP_SFXREVERB dsp in the mix graph. This is unrelated to the
number of possible Reverb3D objects, which is unlimited.

C/C++ Syntax

#define FMOD_REVERB_MAXINSTANCES 4

JavaScript Syntax

FMOD.REVERB_MAXINSTANCES
Values

`FMOD_REVERB_MAXINSTANCES`
See Also

- ChannelControl::setReverbProperties
- ChannelControl::setReverbProperties
- System::setReverbProperties
- System::getReverbProperties
Firelight Technologies FMOD Studio API
**FMOD\_REVERB\_PRESETS**

Sets of predefined reverb properties used to initialize an
**FMOD\_REVERB\_PROPERTIES** structure statically.

i.e. **FMOD\_REVERB\_PROPERTIES** prop = **FMOD\_PRESET\_GENERIC**;

**C/C++ Syntax**

```c
#define FMOD_PRESET\_OFF \{ 1000, \ 7, \ 11, 5000, 100, 100, 100, 250
#define FMOD\_PRESET\_GENERIC \{ 1500, \ 7, \ 11, 5000, 83, 100, 100,
#define FMOD\_PRESET\_PADDEDCELL \{ 170, \ 1, \ 2, 5000, 10, 100, 1
#define FMOD\_PRESET\_ROOM \{ 400, \ 2, \ 3, 5000, 83, 100, 100, 25
#define FMOD\_PRESET\_BATHROOM \{ 1500, \ 7, \ 11, 5000, 54, 100, 60
#define FMOD\_PRESET\_LIVINGROOM \{ 500, \ 3, \ 4, 5000, 10, 100, 1
#define FMOD\_PRESET\_STONEROOM \{ 2300, \ 12, \ 17, 5000, 64, 100, 10
#define FMOD\_PRESET\_ROOM \{ 4300, \ 20, \ 30, 5000, 59, 100, 1
#define FMOD\_PRESET\_CONCERTHALL \{ 3900, \ 20, \ 29, 5000, 70, 100,
#define FMOD\_PRESET\_CAVE \{ 2900, \ 15, \ 22, 5000, 100, 100, 100, 25
#define FMOD\_PRESET\_AREA \{ 7200, \ 20, \ 30, 5000, 33, 100, 100, 2
#define FMOD\_PRESET\_HANGAR \{ 10000, \ 20, \ 30, 5000, 23, 100, 100,
#define FMOD\_PRESET\_CARPET\_HALLWAY \{ 300, \ 2, \ 30, 5000, 10,
#define FMOD\_PRESET\_HALLWAY \{ 1500, \ 7, \ 11, 5000, 59, 100, 100,
#define FMOD\_PRESET\_STONECORRIDOR \{ 270, \ 13, \ 20, 5000, 79, 100
#define FMOD\_PRESET\_ALLEY \{ 1500, \ 7, \ 11, 5000, 86, 100, 100, 2
#define FMOD\_PRESET\_FOREST \{ 1500, \ 162, \ 88, 5000, 54, 79, 100,
#define FMOD\_PRESET\_CITY \{ 1500, \ 7, \ 11, 5000, 67, 50, 100, 25
#define FMOD\_PRESET\_MOUNTAINS \{ 1500, \ 300, 100, 5000, 21, 27, 10
#define FMOD\_PRESET\_QUARRY \{ 1500, \ 61, \ 25, 5000, 83, 100, 100,
#define FMOD\_PRESET\_PLAIN \{ 1500, \ 179, 100, 5000, 50, 21, 100, 2
#define FMOD\_PRESET\_PARKINGLOT \{ 1700, \ 8, \ 12, 5000, 100, 100, 1
#define FMOD\_PRESET\_SEWERPIPE \{ 2800, \ 14, \ 21, 5000, 14, 80, 6
#define FMOD\_PRESET\_UNDERWATER \{ 1500, \ 7, \ 11, 5000, 10, 100, 1
```
Values

*FMOD_PRESET_OFF*
Off / disabled

*FMOD_PRESET_GENERIC*
Generic / default

*FMOD_PRESET_PADDEDCELL*
Padded cell

*FMOD_PRESET_ROOM*
Room

*FMOD_PRESET_BATHROOM*
Bathroom

*FMOD_PRESET_LIVINGROOM*
Living room

*FMOD_PRESET_STONEROOM*
Stone room

*FMOD_PRESET_AUDITORIUM*
Auditorium

*FMOD_PRESET_CONCERTHALL*
Convert hall

*FMOD_PRESET_CAVE*
Cave

_FMOD_PRESET_ARENA_

Arena

_FMOD_PRESET_HANGAR_

Hangar

_FMOD_PRESET_CARPETTEDHALLWAY_

Carpeted hallway

_FMOD_PRESET_HALLWAY_

Hallway

_FMOD_PRESET_STONECORRIDOR_

Stone corridor

_FMOD_PRESET_ALLEY_

Alley

_FMOD_PRESET_FOREST_

Forest

_FMOD_PRESET_CITY_

City

_FMOD_PRESET_MOUNTAINS_

Mountains

_FMOD_PRESET_QUARRY_

Quarry
FMOD_PRESET_PLAIN

Plain

FMOD_PRESET_PARKINGLOT

Parking lot

FMOD_PRESET_SEWERPIPE

Sewer pipe

FMOD_PRESET_UNDERWATER

Underwater
See Also

- System::setReverbProperties
- System::getReverbProperties
Firelight Technologies FMOD Studio API
**FMOD_SYSTEM_CALLBACK_TYP**

These callback types are used with `System::setCallback`.

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_CALLBACK_DEVICELISTCHANGED` (Windows only) will disable any automated device ejection/insertion handling by FMOD. Use this callback to control the behaviour yourself.

**Note!** Using `FMOD_SYSTEM_CALLBACK_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_CALLBACK_MEMORYALLOCATIONFAILED` callback.

**C/C++ Syntax**

```c
#define FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED 0x00000001
#define FMOD_SYSTEM_CALLBACK_DEVICELOST 0x00000002
#define FMOD_SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED 0x00000004
#define FMOD_SYSTEM_CALLBACK_THREADCREATED 0x00000008
#define FMOD_SYSTEM_CALLBACK_BADDSPCONNECTION 0x00000010
#define FMOD_SYSTEM_CALLBACK_PREMIX 0x00000020
#define FMOD_SYSTEM_CALLBACK_POSTMIX 0x00000040
#define FMOD_SYSTEM_CALLBACK_ERROR 0x00000080
#define FMOD_SYSTEM_CALLBACK_MIDMIX 0x00001000
#define FMOD_SYSTEM_CALLBACK_THREADDESTROYED 0x00002000
#define FMOD_SYSTEM_CALLBACK_PREUPDATE 0x00004000
#define FMOD_SYSTEM_CALLBACK_POSTUPDATE 0x00008000
#define FMOD_SYSTEM_CALLBACK_RECORDLISTCHANGED 0x00010000
#define FMOD_SYSTEM_CALLBACK_ALL 0xFFFFFFFF
```

**JavaScript Syntax**

```javascript
```
FMOD.SYSTEM_CALLBACK_DEVICELISTCHANGED
FMOD.SYSTEM_CALLBACK_DEVICELOST
FMOD.SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED
FMOD.SYSTEM_CALLBACK_THREADCREATED
FMOD.SYSTEM_CALLBACK_BADDSPCONNECTION
FMOD.SYSTEM_CALLBACK_PREMIX
FMOD.SYSTEM_CALLBACK_POSTMIX
FMOD.SYSTEM_CALLBACK_ERROR
FMOD.SYSTEM_CALLBACK_MIDMIX
FMOD.SYSTEM_CALLBACK_THREADDESTROYED
FMOD.SYSTEM_CALLBACK_PREUPDATE
FMOD.SYSTEM_CALLBACK_POSTUPDATE
FMOD.SYSTEM_CALLBACK_RECORDLISTCHANGED
FMOD.SYSTEM_CALLBACK_ALL
**Values**

*FMOD_SYSTEM_CALLBACK_DEVICELISTCHANGED*

Called from [System::update](#) when the enumerated list of devices has changed.

*FMOD_SYSTEM_CALLBACK_DEVICELOST*

Called from [System::update](#) when an output device has been lost due to control panel parameter changes and FMOD cannot automatically recover.

*FMOD_SYSTEM_CALLBACK_MEMORYALLOCATIONFAILED*

Called directly when a memory allocation fails somewhere in FMOD. (NOTE - 'system' will be NULL in this callback type.)

*FMOD_SYSTEM_CALLBACK_THREADCREATED*

Called directly when a thread is created.

*FMOD_SYSTEM_CALLBACK_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_CALLBACK_PREMIX*

Called each tick before a mix update happens.

*FMOD_SYSTEM_CALLBACK_POSTMIX*

Called each tick after a mix update happens.

*FMOD_SYSTEM_CALLBACK_ERROR*

Called when each API function returns an error code, including delayed async functions.

*FMOD_SYSTEM_CALLBACK_MIDMIX*
Called each tick in mix update after clocks have been updated before the main mix occurs.

**FMOD_SYSTEM_CALLBACK_THREADDESTROYED**

Called directly when a thread is destroyed.

**FMOD_SYSTEM_CALLBACK_PREUPDATE**

Called at start of System::update function.

**FMOD_SYSTEM_CALLBACK_POSTUPDATE**

Called at end of System::update function.

**FMOD_SYSTEM_CALLBACK_RECORDLISTCHANGED**

Called from System::update when the enumerated list of recording devices has changed.

**FMOD_SYSTEM_CALLBACK_ALL**

Pass this mask to System::setCallback to receive all callback types.
See Also

- System::setCallback
- System::update
- DSP::addInput
Firelight Technologies FMOD Studio API
FMOD_TIMEUNIT

List of time types that can be returned by Sound::getLength and used with Channel::setPosition or Channel::getPosition.

C/C++ Syntax

#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

JavaScript Syntax

FMOD.TIMEUNIT_MS
FMOD.TIMEUNIT_PCM
FMOD.TIMEUNIT_PCMBYTES
FMOD.TIMEUNIT_RAWBYTES
FMOD.TIMEUNIT_PCMFRACTION
FMOD.TIMEUNIT_MODORDER
FMOD.TIMEUNIT_MODROW
FMOD.TIMEUNIT_MODPATTERN
FMOD.TIMEUNIT_BUFFERED
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. Cannot use with Channel::setPosition. 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. Cannot use with Channel::setPosition. Sound::getLength will return the number of patterns
in the song and `Channel::getPosition` will return the currently playing pattern.
See Also

- Sound::getLength
- Channel::setPosition
- Channel::getPosition
Firelight Technologies FMOD Studio API
Enumerations

FMOD_CHANNELCONTROL_CALLBACK_TYPE
FMOD_CHANNELCONTROL_DSP_INDEX
FMOD_CHANNELCONTROL_TYPE
FMOD_CHANNELORDER
FMOD_DEBUG_MODE
FMOD_DSPCONNECTION_TYPE
FMOD_DSP_CHANNELMIX
FMOD_DSP_CHANNELMIX_OUTPUT
FMOD_DSP_CHORUS
FMOD_DSP_COMPRESSOR
FMOD_DSP_CONVOLUTION_REVERB
FMOD_DSP_DELAY
FMOD_DSP_DISTORTION
FMOD_DSP_ECHO
FMOD_DSP_ENVELOPEFOLLOWER
FMOD_DSP_FADER
FMOD_DSP_FFT
FMOD_DSP_FFT_WINDOW
FMOD_DSP_FLANGE
FMOD_DSP_HIGHPASS
FMOD_DSP_HIGHPASS_SIMPLE
FMOD_DSP_ITECHO
FMOD_DSP_ITLOWPASS
FMOD_DSP_LIMITER
FMOD_DSP_LOWPASS
FMOD_DSP_LOWPASS_SIMPLE
FMOD_DSP_MULTIBAND_EQ
FMOD_DSP_MULTIBAND_EQ_FILTER_TYPE
FMOD_DSP_NORMALIZE
FMOD_DSP_OBJECTPAN
FMOD_DSP_OSCILLATOR
FMOD_DSP_PAN
FMOD_DSP_PAN_2D_STEREO_MODE_TYPE
FMOD_DSP_PAN_3D_EXTENT_MODE_TYPE
FMOD_DSP_PAN_3D_ROLLOFF_TYPE
FMOD_DSP_PAN_MODE_TYPE
FMOD_DSP_PAN_SURROUND_FLAGS
FMOD_DSP_PARAMEQ
FMOD_DSP_PARAMETER_DATA_TYPE
FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE
FMOD_DSP_PARAMETER_TYPE
FMOD_DSP_PITCHSHIFT
FMOD_DSP_PROCESS_OPERATION
FMOD_DSP_RESAMPLER
FMOD_DSP_RETURN
FMOD_DSP_SEND
FMOD_DSP_SFXREVERB
FMOD_DSP_THREE_EQ
FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_TYPE
FMOD_DSP_TRANSCEIVER
FMOD_DSP_TRANSCEIVER_SPEAKERMODE
FMOD_DSP_TREMOLO
FMOD_DSP_TYPE
FMOD_ERRORCALLBACK_INSTANCETYPE
FMOD_OPENSTATE
FMOD_OUTPUTTYPE
FMOD_PLUGINTYPE
FMOD_RESULT
FMOD_SOUNDGROUP_BEHAVIOR
FMOD_SOUND_FORMAT
FMOD_SOUND_TYPE
FMOD_SPEAKER
FMOD_SPEAKERMODE
FMOD_TAGDATATYPE
FMOD_TAGTYPE
Firelight Technologies FMOD Studio API
These callback types are used with `Channel::setCallback`.

### C/C++ Syntax

```c
typedef enum {
    FMOD_CHANNELCONTROL_CALLBACK_END,
    FMOD_CHANNELCONTROL_CALLBACK_VIRTUALVOICE,
    FMOD_CHANNELCONTROL_CALLBACK_SYNCPOINT,
    FMOD_CHANNELCONTROL_CALLBACK_OCCLUSION,
    FMOD_CHANNELCONTROL_CALLBACK_MAX
} FMOD_CHANNELCONTROL_CALLBACK_TYPE;
```

### JavaScript Syntax

```javascript
FMOD.CHANNELCONTROL_CALLBACK_END
FMOD.CHANNELCONTROL_CALLBACK_VIRTUALVOICE
FMOD.CHANNELCONTROL_CALLBACK_SYNCPOINT
FMOD.CHANNELCONTROL_CALLBACK_OCCLUSION
FMOD.CHANNELCONTROL_CALLBACK_MAX
FMOD.CHANNELCONTROL_CALLBACK_FORCEINT
```
Values

*FMOD_CHANNELCONTROL_CALLBACK_END*

Called when a sound ends.

*FMOD_CHANNELCONTROL_CALLBACK_VIRTUALVOICE*

Called when a voice is swapped out or swapped in.

*FMOD_CHANNELCONTROL_CALLBACK_SYNCPOINT*

Called when a syncpoint is encountered. Can be from wav file markers.

*FMOD_CHANNELCONTROL_CALLBACK_OCCLUSION*

Called when the channel has its geometry occlusion value calculated. Can be used to clamp or change the value.

*FMOD_CHANNELCONTROL_CALLBACK_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_CHANNELCONTROL_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`
- `ChannelGroup::setCallback`
- `FMOD_CHANNELCONTROL_CALLBACK`
- `System::update`
Firelight Technologies FMOD Studio API
These enums denote special types of node within a DSP chain.

C/C++ Syntax

typedef enum {
    FMOD_CHANNELCONTROL_DSP_HEAD,
    FMOD_CHANNELCONTROL_DSP_FADER,
    FMOD_CHANNELCONTROL_DSP_TAIL
} FMOD_CHANNELCONTROL_DSP_INDEX;

JavaScript Syntax

FMOD.CHANNELCONTROL_DSP_HEAD
FMOD.CHANNELCONTROL_DSP_FADER
FMOD.CHANNELCONTROL_DSP_TAIL
FMOD.CHANNELCONTROL_DSP_FORCEINT
**Values**

*FMOD_CHANNELCONTROL_DSP_HEAD*

Head of the DSP chain. Equivalent of index 0.

*FMOD_CHANNELCONTROL_DSP_FADER*

Built in fader DSP.

*FMOD_CHANNELCONTROL_DSP_TAIL*

Tail of the DSP chain. Equivalent of the number of dsps minus 1.
Remarks

By default there is 1 fader for a ChannelGroup or Channel, and it is the head.
See Also

- `Channel::getDSP`
- `ChannelGroup::getDSP`
- `ChannelControl::getNumDSPs`
- `ChannelControl::setDSPIndex`
Firelight Technologies FMOD Studio API
FMOD_CHANNELCONTROL_TYPE

Used to distinguish if a FMOD_CHANNELCONTROL parameter is actually a channel or a channelgroup.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_CHANNELCONTROL_CHANNEL,
    FMOD_CHANNELCONTROL_CHANNELGROUP
} FMOD_CHANNELCONTROL_TYPE;
```

**JavaScript Syntax**

```javascript
FMOD.CHANNELCONTROL_CHANNEL
FMOD.CHANNELCONTROL_CHANNELGROUP
FMOD.CHANNELCONTROL_FORCEINT
```
Values

`FMOD_CHANNELCONTROL_CHANNEL`

`FMOD_CHANNELCONTROL_CHANNELGROUP`
Remarks

Cast the FMOD_CHANNELCONTROL to an FMOD_CHANNEL/FMOD::Channel, or FMOD_CHANNELGROUP/FMOD::ChannelGroup if specific functionality is needed for either class. Otherwise use as FMOD_CHANNELCONTROL/FMOD::ChannelControl and use that API.
See Also

- Channel::setCallback
- ChannelGroup::setCallback
Firelight Technologies FMOD Studio API
FMOD_CHANNELORDER

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.

C/C++ Syntax

typedef enum {
    FMOD_CHANNELORDER_DEFAULT,
    FMOD_CHANNELORDER_WAVEFORMAT,
    FMOD_CHANNELORDER_PROTOOLS,
    FMOD_CHANNELORDER_ALLMONO,
    FMOD_CHANNELORDER_ALLSTEREO,
    FMOD_CHANNELORDER_ALSA,
    FMOD_CHANNELORDER_MAX
} FMOD_CHANNELORDER;

JavaScript Syntax

FMOD.CHANNELORDER_DEFAULT
FMOD.CHANNELORDER_WAVEFORMAT
FMOD.CHANNELORDER_PROTOOLS
FMOD.CHANNELORDER_ALLMONO
FMOD.CHANNELORDER_ALLSTEREO
FMOD.CHANNELORDER_ALSA
FMOD.CHANNELORDER_MAX
FMOD.CHANNELORDER_FORCEINT
Values

**FMOD_CHANNELORDER_DEFAULT**
Left, Right, Center, LFE, Surround Left, Surround Right, Back Left, Back Right (see [FMOD_SPEAKER](#) enumeration)

**FMOD_CHANNELORDER_WAVEFORMAT**
Left, Right, Center, LFE, Back Left, Back Right, Surround Left, Surround Right (as per Microsoft .wav WAVEFORMAT structure master order)

**FMOD_CHANNELORDER_PROTOOLS**
Left, Center, Right, Surround Left, Surround Right, LFE

**FMOD_CHANNELORDER_ALLMONO**
Mono, Mono, Mono, Mono, Mono, Mono, ... (each channel all the way up to [FMOD_MAX_CHANNEL_WIDTH](#) channels are treated as if they were mono)

**FMOD_CHANNELORDER_ALLSTEREO**
Left, Right, Left, Right, Left, Right, ... (each pair of channels is treated as stereo all the way up to [FMOD_MAX_CHANNEL_WIDTH](#) channels)

**FMOD_CHANNELORDER_ALSA**
Left, Right, Surround Left, Surround Right, Center, LFE (as per Linux ALSA channel order)

**FMOD_CHANNELORDER_MAX**
Maximum number of channel orderings supported.
See Also

- FMOD_CREATENSITYINFO
- FMOD_MAX_CHANNEL_WIDTH
Firelight Technologies FMOD Studio API
FMOD_DEBUG_MODE

Specify the destination of log output when using the logging version of FMOD.

C/C++ Syntax

typedef enum {
    FMOD_DEBUG_MODE_TTY,
    FMOD_DEBUG_MODE_FILE,
    FMOD_DEBUG_MODE_CALLBACK
} FMOD_DEBUG_MODE;

JavaScript Syntax

FMOD.DEBUG_MODE_TTY
FMOD.DEBUG_MODE_FILE
FMOD.DEBUG_MODE_CALLBACK
FMOD.DEBUG_MODE_FORCEINT
Values

**FMOD_DEBUG_MODE_TTY**

Default log location per platform, i.e. Visual Studio output window, stderr, LogCat, etc

**FMOD_DEBUG_MODE_FILE**

Write log to specified file path

**FMOD_DEBUG_MODE_CALLBACK**

Call specified callback with log information
Remarks

TTY destination can vary depending on platform, common examples include the Visual Studio / Xcode output window, stderr and LogCat.
See Also

- FMOD_Debug_Initialize
Firelight Technologies FMOD Studio API
FMOD_DSPCONNECTION_TYPE

List of connection types between 2 DSP nodes.

C/C++ Syntax

typedef enum {
    FMOD_DSPCONNECTION_TYPE_STANDARD,
    FMOD_DSPCONNECTION_TYPE_SIDECHAIN,
    FMOD_DSPCONNECTION_TYPE_SEND,
    FMOD_DSPCONNECTION_TYPE_SEND_SIDECHAIN,
    FMOD_DSPCONNECTION_TYPE_MAX
} FMOD_DSPCONNECTION_TYPE;

JavaScript Syntax

FMOD_DSPCONNECTION_TYPE_STANDARD
FMOD_DSPCONNECTION_TYPE_SIDECHAIN
FMOD_DSPCONNECTION_TYPE_SEND
FMOD_DSPCONNECTION_TYPE_SEND_SIDECHAIN
FMOD_DSPCONNECTION_TYPE_MAX
FMOD_DSPCONNECTION_TYPE_FORCEINT
Values

**FMOD_DSPCONNECTION_TYPE_STANDARD**
Default connection type. Audio is mixed from the input to the output DSP's audible buffer.

**FMOD_DSPCONNECTION_TYPE_SIDECHAIN**
Sidechain connection type. Audio is mixed from the input to the output DSP's sidechain buffer.

**FMOD_DSPCONNECTION_TYPE_SEND**
Send connection type. Audio is mixed from the input to the output DSP's audible buffer, but the input is NOT executed, only copied from. A standard connection or sidechain needs to make an input execute to generate data.

**FMOD_DSPCONNECTION_TYPE_SEND_SIDECHAIN**
Send sidechain connection type. Audio is mixed from the input to the output DSP's sidechain buffer, but the input is NOT executed, only copied from. A standard connection or sidechain needs to make an input execute to generate data.

**FMOD_DSPCONNECTION_TYPE_MAX**
Maximum number of DSP connection types supported.
Remarks

FMOD_DSP_CONNECTION_TYPE_STANDARD
----------------------------------
Default DSPConnection type. Audio is mixed from the input to the output DSP's audible buffer, meaning it will be part of the audible signal. A standard connection will execute its input DSP if it has not been executed before.

FMOD_DSP_CONNECTION_TYPE_SIDECHAIN
----------------------------------
Sidechain DSPConnection type. Audio is mixed from the input to the output DSP's sidechain buffer, meaning it will NOT be part of the audible signal. A sidechain connection will execute its input DSP if it has not been executed before.
The purpose of the separate sidechain buffer in a DSP, is so that the DSP effect can privately access for analysis purposes. An example of use in this case, could be a compressor which analyzes the signal, to control its own effect parameters (ie a compression level or gain).

For the effect developer, to accept sidechain data, the sidechain data will appear in the **FMOD_DSP_STATE** struct which is passed into the read callback of a DSP unit. **FMOD_DSP_STATE::sidechaindata** and **FMOD_DSP::sidechainchannels** will hold the mixed result of any sidechain data flowing into it.

FMOD_DSP_CONNECTION_TYPE_SEND
-----------------------------
Send DSPConnection type. Audio is mixed from the input to the output DSP's audible buffer, meaning it will be part of the audible signal. A send connection will NOT execute its input DSP if it has not been executed before. A send connection will only read what exists at the input's buffer at the time of executing the output DSP unit (which can be considered the 'return')

FMOD_DSP_CONNECTION_TYPE_SEND_SIDECHAIN
---------------------------------------
Send sidechain DSPConnection type. Audio is mixed from the input to the output DSP's sidechain buffer, meaning it will NOT be part of the audible signal.
A send sidechain connection will NOT execute its input DSP if it has not been executed before. A send sidechain connection will only read what exists at the input's buffer at the time of executing the output DSP unit (which can be considered the 'sidechain return').
For the effect developer, to accept sidechain data, the sidechain data will appear in the FMOD_DSP_STATE struct which is passed into the read callback of a DSP unit. FMOD_DSP_STATE::sidechaindata and FMOD_DSP::sidechainchannels will hold the mixed result of any sidechain data flowing into it.
See Also

- DSP::addInput
- DSPConnection::getType
Firelight Technologies FMOD Studio API
FMOD_DSP_CHANNELMIX

Parameter types for the FMOD_DSP_TYPE_CHANNELMIX filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_CHANNELMIX_OUTPUTGROUPING,
    FMOD_DSP_CHANNELMIX_GAIN_CH0,
    FMOD_DSP_CHANNELMIX_GAIN_CH1,
    FMOD_DSP_CHANNELMIX_GAIN_CH2,
    FMOD_DSP_CHANNELMIX_GAIN_CH3,
    FMOD_DSP_CHANNELMIX_GAIN_CH4,
    FMOD_DSP_CHANNELMIX_GAIN_CH5,
    FMOD_DSP_CHANNELMIX_GAIN_CH6,
    FMOD_DSP_CHANNELMIX_GAIN_CH7,
    FMOD_DSP_CHANNELMIX_GAIN_CH8,
    FMOD_DSP_CHANNELMIX_GAIN_CH9,
    FMOD_DSP_CHANNELMIX_GAIN_CH10,
    FMOD_DSP_CHANNELMIX_GAIN_CH11,
    FMOD_DSP_CHANNELMIX_GAIN_CH12,
    FMOD_DSP_CHANNELMIX_GAIN_CH13,
    FMOD_DSP_CHANNELMIX_GAIN_CH14,
    FMOD_DSP_CHANNELMIX_GAIN_CH15,
    FMOD_DSP_CHANNELMIX_GAIN_CH16,
    FMOD_DSP_CHANNELMIX_GAIN_CH17,
    FMOD_DSP_CHANNELMIX_GAIN_CH18,
    FMOD_DSP_CHANNELMIX_GAIN_CH19,
    FMOD_DSP_CHANNELMIX_GAIN_CH20,
    FMOD_DSP_CHANNELMIX_GAIN_CH21,
    FMOD_DSP_CHANNELMIX_GAIN_CH22,
    FMOD_DSP_CHANNELMIX_GAIN_CH23,
    FMOD_DSP_CHANNELMIX_GAIN_CH24,
    FMOD_DSP_CHANNELMIX_GAIN_CH25,
    FMOD_DSP_CHANNELMIX_GAIN_CH26,
    FMOD_DSP_CHANNELMIX_GAIN_CH27,
    FMOD_DSP_CHANNELMIX_GAIN_CH28,
    FMOD_DSP_CHANNELMIX_GAIN_CH29,
    FMOD_DSP_CHANNELMIX_GAIN_CH30,
    FMOD_DSP_CHANNELMIX_GAIN_CH31
} FMOD_DSP_CHANNELMIX;

JavaScript Syntax
Values

`FMOD_DSP_CHANNELMIX_OUTPUTGROUPING`

(Type:int) - Refer to `FMOD_DSP_CHANNELMIX_OUTPUT` enumeration. Default = `FMOD_DSP_CHANNELMIX_OUTPUT_DEFAULT`. See remarks.

`FMOD_DSP_CHANNELMIX_GAIN_CH0`

(Type:float) - Channel #0 gain in dB. -80.0 to 10.0. Default = 0.

`FMOD_DSP_CHANNELMIX_GAIN_CH1`

(Type:float) - Channel #1 gain in dB. -80.0 to 10.0. Default = 0.

`FMOD_DSP_CHANNELMIX_GAIN_CH2`

(Type:float) - Channel #2 gain in dB. -80.0 to 10.0. Default = 0.

`FMOD_DSP_CHANNELMIX_GAIN_CH3`

(Type:float) - Channel #3 gain in dB. -80.0 to 10.0. Default = 0.

`FMOD_DSP_CHANNELMIX_GAIN_CH4`

(Type:float) - Channel #4 gain in dB. -80.0 to 10.0. Default = 0.

`FMOD_DSP_CHANNELMIX_GAIN_CH5`

(Type:float) - Channel #5 gain in dB. -80.0 to 10.0. Default = 0.

`FMOD_DSP_CHANNELMIX_GAIN_CH6`

(Type:float) - Channel #6 gain in dB. -80.0 to 10.0. Default = 0.

`FMOD_DSP_CHANNELMIX_GAIN_CH7`

(Type:float) - Channel #7 gain in dB. -80.0 to 10.0. Default = 0.
**FMOD_DSP_CHANNELMIX_GAIN_CH8**

(Type: float) - Channel #8 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH9**

(Type: float) - Channel #9 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH10**

(Type: float) - Channel #10 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH11**

(Type: float) - Channel #11 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH12**

(Type: float) - Channel #12 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH13**

(Type: float) - Channel #13 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH14**

(Type: float) - Channel #14 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH15**

(Type: float) - Channel #15 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH16**

(Type: float) - Channel #16 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH17**

(Type: float) - Channel #17 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH18**

(Type: float) - Channel #18 gain in dB. -80.0 to 10.0. Default = 0.
(Type: float) - Channel #18 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH19

(Type: float) - Channel #19 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH20

(Type: float) - Channel #20 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH21

(Type: float) - Channel #21 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH22

(Type: float) - Channel #22 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH23

(Type: float) - Channel #23 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH24

(Type: float) - Channel #24 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH25

(Type: float) - Channel #25 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH26

(Type: float) - Channel #26 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH27

(Type: float) - Channel #27 gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_CHANNELMIX_GAIN_CH28

(Type: float) - Channel #28 gain in dB. -80.0 to 10.0. Default = 0.
**FMOD_DSP_CHANNELMIX_GAIN_CH29**

(Type: float) - Channel #29 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH30**

(Type: float) - Channel #30 gain in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CHANNELMIX_GAIN_CH31**

(Type: float) - Channel #31 gain in dB. -80.0 to 10.0. Default = 0.
Remarks

For **FMOD_DSP_CHANNELMIX_OUTPUTGROUPING**, this value will set the output speaker format for the DSP, and also map the incoming channels to the outgoing channels in a round-robin fashion. Use this for example play a 32 channel input signal as if it were a repeating group of output signals. Ie.  
**FMOD_DSP_CHANNELMIX_OUTPUT_ALLMONO** = all incoming channels are mixed to a mono output.  
**FMOD_DSP_CHANNELMIX_OUTPUT_ALLSTEREO** = all incoming channels are mixed to a stereo output, ie even incoming channels 0,2,4,6,etc are mixed to left, and odd incoming channels 1,3,5,7,etc are mixed to right.  
**FMOD_DSP_CHANNELMIX_OUTPUT_ALL5POINT1** = all incoming channels are mixed to a 5.1 output. If there are less than 6 coming in, it will just fill the first n channels in the 6 output channels. If there are more, then it will repeat the input pattern to the output like it did with the stereo case, ie 12 incoming channels are mapped as 0-5 mixed to the 5.1 output and 6 to 11 mapped to the 5.1 output.  
**FMOD_DSP_CHANNELMIX_OUTPUT_ALLLFE** = all incoming channels are mixed to a 5.1 output but via the LFE channel only.
See Also

- DSP::setParameterInt
- DSP::getParameterInt
- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
Parameter types for the **FMOD_DSP_CHANNELMIX_OUTPUTGROUPING** parameter for **FMOD_DSP_TYPE_CHANNELMIX** effect.

### C/C++ Syntax

typedef enum {
    FMOD_DSP_CHANNELMIX_OUTPUT_DEFAULT,
    FMOD_DSP_CHANNELMIX_OUTPUT_ALLMONO,
    FMOD_DSP_CHANNELMIX_OUTPUT_ALLSTEREO,
    FMOD_DSP_CHANNELMIX_OUTPUT_ALLQUAD,
    FMOD_DSP_CHANNELMIX_OUTPUT_ALL5POINT1,
    FMOD_DSP_CHANNELMIX_OUTPUT_ALL7POINT1,
    FMOD_DSP_CHANNELMIX_OUTPUT_ALLLFE
} FMOD_DSP_CHANNELMIX_OUTPUT;

### JavaScript Syntax

```javascript
FMOD.DSP_CHANNELMIX_OUTPUT_DEFAULT
FMOD.DSP_CHANNELMIX_OUTPUT_ALLMONO
FMOD.DSP_CHANNELMIX_OUTPUT_ALLSTEREO
FMOD.DSP_CHANNELMIX_OUTPUT_ALLQUAD
FMOD.DSP_CHANNELMIX_OUTPUT_ALL5POINT1
FMOD.DSP_CHANNELMIX_OUTPUT_ALL7POINT1
FMOD.DSP_CHANNELMIX_OUTPUT_ALLLFE
```
Values

*FMOD_DSP_CHANNELMIX_OUTPUT_DEFAULT*

Output channel count = input channel count. Mapping: See *FMOD_SPEAKER* enumeration.

*FMOD_DSP_CHANNELMIX_OUTPUT_ALLMONO*

Output channel count = 1. Mapping: Mono, Mono, Mono, Mono, Mono, Mono, ... (each channel all the way up to *FMOD_MAX_CHANNEL_WIDTH* channels are treated as if they were mono)

*FMOD_DSP_CHANNELMIX_OUTPUT_ALLSTEREO*

Output channel count = 2. Mapping: Left, Right, Left, Right, Left, Right, ... (each pair of channels is treated as stereo all the way up to *FMOD_MAX_CHANNEL_WIDTH* channels)

*FMOD_DSP_CHANNELMIX_OUTPUT_ALLQUAD*


*FMOD_DSP_CHANNELMIX_OUTPUT_ALL5POINT1*

Output channel count = 6. Mapping: Repeating pattern of Front Left, Front Right, Center, LFE, Surround Left, Surround Right.

*FMOD_DSP_CHANNELMIX_OUTPUT_ALL7POINT1*

Output channel count = 8. Mapping: Repeating pattern of Front Left, Front Right, Center, LFE, Surround Left, Surround Right, Back Left, Back Right.

*FMOD_DSP_CHANNELMIX_OUTPUT_ALLLFE*

See Also

- DSP::setParameterInt
- DSP::getParameterInt
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_CHORUS

Parameter types for the FMOD_DSP_TYPE_CHORUS filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_CHORUS_MIX,
    FMOD_DSP_CHORUS_RATE,
    FMOD_DSP_CHORUS_DEPTH
} FMOD_DSP_CHORUS;

JavaScript Syntax

FMOD.DSP_CHORUS_MIX
FMOD.DSP_CHORUS_RATE
FMOD.DSP_CHORUS_DEPTH
Values

**FMOD_DSP_CHORUS_MIX**

(Type: float) - Volume of original signal to pass to output. 0.0 to 100.0. Default = 50.0.

**FMOD_DSP_CHORUS_RATE**

(Type: float) - Chorus modulation rate in Hz. 0.0 to 20.0. Default = 0.8 Hz.

**FMOD_DSP_CHORUS_DEPTH**

(Type: float) - Chorus modulation depth. 0.0 to 100.0. Default = 3.0.
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::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_COMPRESSOR

Parameter types for the FMOD_DSP_TYPE_COMPRESSOR unit. This is a multichannel software limiter that is uniform across the whole spectrum.

C/C++ Syntax

typedef enum {
   FMOD_DSP_COMPRESSOR_THRESHOLD,
   FMOD_DSP_COMPRESSOR_RATIO,
   FMOD_DSP_COMPRESSOR_ATTACK,
   FMOD_DSP_COMPRESSOR_RELEASE,
   FMOD_DSP_COMPRESSOR_GAINMAKEUP,
   FMOD_DSP_COMPRESSOR_USESIDECHAIN,
   FMOD_DSP_COMPRESSOR_LINKED
} FMOD_DSP_COMPRESSOR;

JavaScript Syntax

FMOD.DSP_COMPRESSOR_THRESHOLD
FMOD.DSP_COMPRESSOR_RATIO
FMOD.DSP_COMPRESSOR_ATTACK
FMOD.DSP_COMPRESSOR_RELEASE
FMOD.DSP_COMPRESSOR_GAINMAKEUP
FMOD.DSP_COMPRESSOR_USESIDECHAIN
FMOD.DSP_COMPRESSOR_LINKED
Values

**FMOD_DSP_COMPRESSOR_THRESHOLD**

(Type: float) - Threshold level (dB) in the range from -80 through 0. Default = 0.

**FMOD_DSP_COMPRESSOR_RATIO**

(Type: float) - Compression Ratio (dB/dB) in the range from 1 to 50. Default = 2.5.

**FMOD_DSP_COMPRESSOR_ATTACK**

(Type: float) - Attack time (milliseconds), in the range from 0.1 through 1000. Default value is 20.

**FMOD_DSP_COMPRESSOR_RELEASE**

(Type: float) - Release time (milliseconds), in the range from 10 through 5000. Default value is 100.

**FMOD_DSP_COMPRESSOR_GAINMAKEUP**

(Type: float) - Make-up gain (dB) applied after limiting, in the range from 0 through 30. Default = 0.

**FMOD_DSP_COMPRESSOR_USESIDECHAIN**

(Type: data) - Data of type **FMOD_DSP_PARAMETER_SIDECHAIN**. Whether to analyse the sidechain signal instead of the input signal. Default is { false }

**FMOD_DSP_COMPRESSOR_LINKED**

(Type: bool) - FALSE = Independent (compressor per channel), TRUE = Linked. Default = TRUE.
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::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterBool
- DSP::getParameterBool
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_CONVOLUTION_REVERB

Parameter types for the FMOD_DSP_TYPE_CONVOLUTIONREVERB filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_CONVOLUTION_REVERB_PARAM_IR,
    FMOD_DSP_CONVOLUTION_REVERB_PARAM_WET,
    FMOD_DSP_CONVOLUTION_REVERB_PARAM_DRY,
    FMOD_DSP_CONVOLUTION_REVERB_PARAM_LINKED
} FMOD_DSP_CONVOLUTION_REVERB;

JavaScript Syntax

FMOD.DSP_CONVOLUTION_REVERB_PARAM_IR
FMOD.DSP_CONVOLUTION_REVERB_PARAM_WET
FMOD.DSP_CONVOLUTION_REVERB_PARAM_DRY
FMOD.DSP_CONVOLUTION_REVERB_PARAM_LINKED
Values

**FMOD_DSP_CONVOLUTION_REVERB_PARAM_IR**

(Type:data) - [w] Array of signed 16-bit (short) PCM data to be used as reverb IR. First member of the array should be a 16 bit value (short) which specifies the number of channels. Array looks like [index 0=numchannels][index 1+ = raw 16 bit PCM data]. Data is copied internally so source can be freed.

**FMOD_DSP_CONVOLUTION_REVERB_PARAM_WET**

(Type:float) - [r/w] Volume of echo signal to pass to output in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CONVOLUTION_REVERB_PARAM_DRY**

(Type:float) - [r/w] Original sound volume in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_CONVOLUTION_REVERB_PARAM_LINKED**

(Type:bool) - [r/w] Linked - channels are mixed together before processing through the reverb. Default = TRUE.
Remarks

Convolution Reverb reverb IR.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterData
- DSP::getParameterData
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
**FMOD_DSP_DELAY**

Parameter types for the **FMOD_DSP_TYPE_DELAY** filter.

**C/C++ Syntax**

```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_MAXDELAY
} FMOD_DSP_DELAY;
```

**JavaScript Syntax**

```javascript
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_MAXDELAY
Values

FMOD_DSP_DELAY_CH0
(Type: float) - Channel #0 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH1
(Type: float) - Channel #1 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH2
(Type: float) - Channel #2 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH3
(Type: float) - Channel #3 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH4
(Type: float) - Channel #4 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH5
(Type: float) - Channel #5 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH6
(Type: float) - Channel #6 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH7
(Type: float) - Channel #7 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH8
(Type: float) - Channel #8 Delay in ms. 0 to 10000. Default = 0.

FMOD_DSP_DELAY_CH9
(Type:float) - Channel #9 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH10**

(Type:float) - Channel #10 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH11**

(Type:float) - Channel #11 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH12**

(Type:float) - Channel #12 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH13**

(Type:float) - Channel #13 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH14**

(Type:float) - Channel #14 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_CH15**

(Type:float) - Channel #15 Delay in ms. 0 to 10000. Default = 0.

**FMOD_DSP_DELAY_MAXDELAY**

(Type:float) - 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.

NOTE! Not supported on PlayStation 3.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
Parameter types for the `FMOD_DSP_TYPE_DISTORTION` filter.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_DISTORTION_LEVEL
} FMOD_DSP_DISTORTION;
```

**JavaScript Syntax**

`FMOD.DSP_DISTORTION_LEVEL`
Values

*FMOD_DSP_DISTORTION_LEVEL*

(Type: float) - Distortion value. 0.0 to 1.0. Default = 0.5.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_ECHO

Parameter types for the FMOD_DSP_TYPE_ECHO filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_ECHO_DELAY,
    FMOD_DSP_ECHO_FEEDBACK,
    FMOD_DSP_ECHO_DRYLEVEL,
    FMOD_DSP_ECHO_WETLEVEL
} FMOD_DSP_ECHO;

JavaScript Syntax

FMOD_DSP_ECHO_DELAY
FMOD_DSP_ECHO_FEEDBACK
FMOD_DSP_ECHO_DRYLEVEL
FMOD_DSP_ECHO_WETLEVEL
**Values**

*FMOD_DSP_ECHO_DELAY*

(Type: float) - Echo delay in ms. 10 to 5000. Default = 500.

*FMOD_DSP_ECHO_FEEDBACK*

(Type: float) - Echo decay per delay. 0 to 100. 100.0 = No decay, 0.0 = total decay (ie simple 1 line delay). Default = 50.0.

*FMOD_DSP_ECHO_DRYLEVEL*

(Type: float) - Original sound volume in dB. -80.0 to 10.0. Default = 0.

*FMOD_DSP_ECHO_WETLEVEL*

(Type: float) - Volume of echo signal to pass to output in dB. -80.0 to 10.0. Default = 0.
Remarks

Note. Every time the delay is changed, the plugin re-allocates the echo buffer. This means the echo will disappear at that time while it refills its new buffer. Larger echo delays result in larger amounts of memory allocated.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_ENVELOPEFOLLOWER

Deprecated and will be removed in a future release.

Parameter types for the FMOD_DSP_TYPE_ENVELOPEFOLLOWER unit. This is a simple envelope follower for tracking the signal level.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_ENVELOPEFOLLOWER_ATTACK,
    FMOD_DSP_ENVELOPEFOLLOWER_RELEASE,
    FMOD_DSP_ENVELOPEFOLLOWER_ENVELOPE,
    FMOD_DSP_ENVELOPEFOLLOWER_USESIDECHAIN
} FMOD_DSP_ENVELOPEFOLLOWER;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_ENVELOPEFOLLOWER_ATTACK
FMOD.DSP_ENVELOPEFOLLOWER_RELEASE
FMOD.DSP_ENVELOPEFOLLOWER_ENVELOPE
FMOD.DSP_ENVELOPEFOLLOWER_USESIDECHAIN
```
Values

*FMOD_DSP_ENVELOPE_FOLLOWER_ATTACK*

(Type: float) [r/w] - Attack time (milliseconds), in the range from 0.1 through 1000. Default = 20.

*FMOD_DSP_ENVELOPE_FOLLOWER_RELEASE*

(Type: float) [r/w] - Release time (milliseconds), in the range from 10 through 5000. Default = 100

*FMOD_DSP_ENVELOPE_FOLLOWER_ENVELOPE*

(Type: float) [r] - Current value of the envelope, in the range 0 to 1. Read-only.

*FMOD_DSP_ENVELOPE_FOLLOWER_USESIDECHAIN*

(Type: data) [r/w] - Data of type **FMOD_DSP_PARAMETER_SIDECHAIN**. Whether to analyse the sidechain signal instead of the input signal. Default is `{false}`
Remarks

Deprecated and will be removed in a future release.

This unit does not affect the incoming signal.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterData
- DSP::getParameterData
- FMOD_DSP_TYPE
FMOD_DSP_FADER

Parameter types for the FMOD_DSP_TYPE_FADER filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_FADER_GAIN
} FMOD_DSP_FADER;
Values

\textit{FMOD\_DSP\_FADE\_GAIN}

(Type: float) - Signal gain in dB. -80.0 to 10.0. Default = 0.0.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
**FMOD_DSP_FFT**

Parameter types for the **FMOD_DSP_TYPE_FFT** dsp effect.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_FFT_WINDOWSIZE,
    FMOD_DSP_FFT_WINDOWTYPE,
    FMOD_DSP_FFT_SPECTRUMDATA,
    FMOD_DSP_FFT_DOMINANT_FREQ
} FMOD_DSP_FFT;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_FFT_WINDOWSIZE
FMOD.DSP_FFT_WINDOWTYPE
FMOD.DSP_FFT_SPECTRUMDATA
FMOD.DSP_FFT_DOMINANT_FREQ
```
Values

**FMOD_DSP_FFT_WINDOWSIZE**

(Type:int) - [r/w] Must be a power of 2 between 128 and 16384. 128, 256, 512, 1024, 2048, 4096, 8192, 16384 are accepted. Default = 2048.

**FMOD_DSP_FFT_WINDOWTYPE**

(Type:int) - [r/w] Refer to `FMOD_DSP_FFT_WINDOW` enumeration. Default = `FMOD_DSP_FFT_WINDOW_HAMMING`.

**FMOD_DSP_FFT_SPECTRUMDATA**

(Type:data) - [r] Returns the current spectrum values between 0 and 1 for each 'fft bin'. Cast data to `FMOD_DSP_PARAMETER_DATA_TYPE_FFT`. Divide the niquist rate by the window size to get the hz value per entry.

**FMOD_DSP_FFT_DOMINANT_FREQ**

(Type:float) - [r] Returns the dominant frequencies for each channel.
Remarks

Set the attributes for the spectrum analysis with \texttt{FMOD\_DSP\_FFT\_WINDOWSIZE} and \texttt{FMOD\_DSP\_FFT\_WINDOWTYPE}, and retrieve the results with \texttt{FMOD\_DSP\_FFT\_SPECTRUM} and \texttt{FMOD\_DSP\_FFT\_DOMINANT\_FREQ}. \texttt{FMOD\_DSP\_FFT\_SPECTRUM} stores its data in the \texttt{FMOD\_DSP\_PARAMETER\_DATA\_TYPE\_FFT}. You will need to cast to this structure to get the right data.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterInt
- DSP::getParameterInt
- DSP::setParameterData
- DSP::getParameterData
- FMOD_DSP_TYPE
- FMOD_DSP_FFT_WINDOW
Firelight Technologies FMOD Studio API
List of windowing methods for the **FMOD_DSP_TYPE_FFT** unit. 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.

### C/C++ Syntax

```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;
```

### JavaScript Syntax

```javascript
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
```
**Values**

*FMOD_DSP_FFT_WINDOW_RECT*

\[ w[n] = 1.0 \]

*FMOD_DSP_FFT_WINDOW_TRIANGLE*

\[ w[n] = \text{TRI}(2n/N) \]

*FMOD_DSP_FFT_WINDOW_HAMMING*

\[ w[n] = 0.54 - (0.46 \times \text{COS}(n/N)) \]

*FMOD_DSP_FFT_WINDOW_HANNING*

\[ w[n] = 0.5 \times (1.0 - \text{COS}(n/N)) \]

*FMOD_DSP_FFT_WINDOW_BLACKMAN*

\[ w[n] = 0.42 - (0.5 \times \text{COS}(n/N)) + (0.08 \times \text{COS}(2.0 \times n/N)) \]

*FMOD_DSP_FFT_WINDOW_BLACKMANHARRIS*

\[ w[n] = 0.35875 - (0.48829 \times \text{COS}(1.0 \times n/N)) + (0.14128 \times \text{COS}(2.0 \times n/N)) - (0.01168 \times \text{COS}(3.0 \times n/N)) \]
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**.
- **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.
See Also

- FMOD_DSP_FFT
Firelight Technologies FMOD Studio API
**FMOD_DSP_FLANGE**

Parameter types for the **FMOD_DSP_TYPE_FLANGE** filter.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_FLANGE_MIX,
    FMOD_DSP_FLANGE_DEPTH,
    FMOD_DSP_FLANGE_RATE
} FMOD_DSP_FLANGE;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_FLANGE_MIX
FMOD.DSP_FLANGE_DEPTH
FMOD.DSP_FLANGE_RATE
```
Values

**FMOD_DSP_FLANGE_MIX**

(Type: float) - Percentage of wet signal in mix. 0 to 100. Default = 50.

**FMOD_DSP_FLANGE_DEPTH**

(Type: float) - Flange depth (percentage of 40ms delay). 0.01 to 1.0. Default = 1.0.

**FMOD_DSP_FLANGE_RATE**

(Type: float) - 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::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_HIGHPASS

Parameter types for the FMOD_DSP_TYPE_HIGHPASS filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_HIGHPASS_CUTOFF,
    FMOD_DSP_HIGHPASS_RESONANCE
} FMOD_DSP_HIGHPASS;

JavaScript Syntax

FMOD.DSP_HIGHPASS_CUTOFF
FMOD.DSP_HIGHPASS_RESONANCE
Values

*FMOD_DSP_HIGHPASS_CUTOFF*

(Type: float) - Highpass cutoff frequency in hz. 1.0 to output 22000.0. Default = 5000.0.

*FMOD_DSP_HIGHPASS_RESONANCE*

(Type: float) - Highpass resonance Q value. 1.0 to 10.0. Default = 1.0.
Remarks

Deprecated and will be removed in a future release, to emulate with

```
FMOD_DSP_TYPE_MULTIBAND_EQ:
```

```
// Configure a single band (band A) as a highpass (all other bands defaults to off).
// 12dB rolloff to approximate the old effect curve.
// Cutoff frequency can be used the same as with the old effect.
// Resonance can be applied by setting the 'Q' value of the new effect

FMOD_DSP_SetParameterInt(multiband, FMOD_DSP_MULTIBAND_EQ_A_FILTER,
                        FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_FREQUENCY,
                        FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_Q, res
```
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_HIGHPASS_SIMPLE

Parameter types for the \texttt{FMOD_DSP_TYPE_HIGHPASS_SIMPLE} filter.

\textbf{C/C++ Syntax}

\begin{verbatim}
typedef enum {
    FMOD_DSP_HIGHPASS_SIMPLE_CUTOFF
} FMOD_DSP_HIGHPASS_SIMPLE;
\end{verbatim}

\textbf{JavaScript Syntax}

\begin{verbatim}
FMOD.DSP_HIGHPASS_SIMPLE_CUTOFF
\end{verbatim}
Values

*FMOD_DSP_HIGHPASS_SIMPLE_CUTOFF*

(Type: float) - Highpass cutoff frequency in hz. 10.0 to 22000.0. Default = 1000.0
Remarks

Deprecated and will be removed in a future release, to emulate with `FMOD_DSP_TYPE_MULTIBAND_EQ`:

```cpp
// Configure a single band (band A) as a highpass (all other bands default to off).
// 12dB rolloff to approximate the old effect curve.
// Cutoff frequency can be used the same as with the old effect.
// Resonance / 'Q' should remain at default 0.707.
FMOD_DSP_SetParameterInt(multiband, FMOD_DSP_MULTIBAND_EQ_A_FILTER,
FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_FREQUENCY
```

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.
See Also

- **DSP::setParameterFloat**
- **DSP::getParameterFloat**
- **FMOD_DSP_TYPE**
Firelight Technologies FMOD Studio API
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!

C/C++ Syntax

typedef enum {
    FMOD_DSP_ITECHO_WETDRYMIX,
    FMOD_DSP_ITECHO_FEEDBACK,
    FMOD_DSP_ITECHO_LEFTDELAY,
    FMOD_DSP_ITECHO_RIGHTDELAY,
    FMOD_DSP_ITECHO_PANDELAY
} FMOD_DSP_ITECHO;

JavaScript Syntax

FMOD_DSP_ITECHO_WETDRYMIX
FMOD_DSP_ITECHO_FEEDBACK
FMOD_DSP_ITECHO_LEFTDELAY
FMOD_DSP_ITECHO_RIGHTDELAY
FMOD_DSP_ITECHO_PANDELAY
Values

*FMOD_DSP_ITECHO_WETDRYMIX*

(Type: float) - Ratio of wet (processed) signal to dry (unprocessed) signal. Must be in the range from 0.0 through 100.0 (all wet). Default = 50.

*FMOD_DSP_ITECHO_FEEDBACK*

(Type: float) - Percentage of output fed back into input, in the range from 0.0 through 100.0. Default = 50.

*FMOD_DSP_ITECHO_LEFTDELAY*

(Type: float) - Delay for left channel, in milliseconds, in the range from 1.0 through 2000.0. Default = 500 ms.

*FMOD_DSP_ITECHO_RIGHTDELAY*

(Type: float) - Delay for right channel, in milliseconds, in the range from 1.0 through 2000.0. Default = 500 ms.

*FMOD_DSP_ITECHO_PANDELAY*

(Type: float) - Value that specifies whether to swap left and right delays with each successive echo. Ranges from 0.0 (equivalent to FALSE) to 1.0 (equivalent to TRUE), meaning no swap. Default = 0. CURRENTLY NOT SUPPORTED.
Remarks

Note. Every time the delay is changed, the plugin re-allocates the echo buffer. This means the echo will disappear 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::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
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 Studio's .IT playback uses this filter.

C/C++ Syntax

typedef enum {
  FMOD_DSP_ITLOWPASS_CUTOFF,
  FMOD_DSP_ITLOWPASS_RESONANCE
} FMOD_DSP_ITLOWPASS;

JavaScript Syntax

FMOD.DSP_ITLOWPASS_CUTOFF
FMOD.DSP_ITLOWPASS_RESONANCE
Values

*FMOD_DSP_ITLOWPASS_CUTOFF*

(Type:float) - Lowpass cutoff frequency in hz. 1.0 to 22000.0. Default = 5000.0/

*FMOD_DSP_ITLOWPASS_RESONANCE*

(Type:float) - 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::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_LIMITER

Parameter types for the FMOD_DSP_TYPE_LIMITER filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_LIMITER_RELEASETIME,
    FMOD_DSP_LIMITER_CEILING,
    FMOD_DSP_LIMITER_MAXIMIZERGAIN,
    FMOD_DSP_LIMITER_MODE
} FMOD_DSP_LIMITER;

JavaScript Syntax

FMOD.DSP_LIMITER_RELEASETIME
FMOD.DSP_LIMITER_CEILING
FMOD.DSP_LIMITER_MAXIMIZERGAIN
FMOD.DSP_LIMITER_MODE
Values

*FMOD_DSP_LIMITER_RELEASETIME*

(Type: float) - Time to ramp the silence to full in ms. 1.0 to 1000.0. Default = 10.0.

*FMOD_DSP_LIMITER_CEILING*

(Type: float) - Maximum level of the output signal in dB. -12.0 to 0.0. Default = 0.0.

*FMOD_DSP_LIMITER_MAXIMIZERGAIN*

(Type: float) - Maximum amplification allowed in dB. 0.0 to 12.0. Default = 0.0. 0.0 = no amplification, higher values allow more boost.

*FMOD_DSP_LIMITER_MODE*

(Type: float) - Channel processing mode. 0 or 1. Default = 0. 0 = Independent (limiter per channel), 1 = Linked.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_LOWPASS

Parameter types for the FMOD_DSP_TYPE_LOWPASS filter.

C/C++ Syntax

typedef enum {
   FMOD_DSP_LOWPASS_CUTOFF,
   FMOD_DSP_LOWPASS_RESONANCE
} FMOD_DSP_LOWPASS;

JavaScript Syntax

FMOD.DSP_LOWPASS_CUTOFF
FMOD.DSP_LOWPASS_RESONANCE
Values

_FMOD_DSP_LOWPASS_CUTOFF_

(Type:float) - Lowpass cutoff frequency in hz. 10.0 to 22000.0. Default = 5000.0.

_FMOD_DSP_LOWPASS_RESONANCE_

(Type:float) - Lowpass resonance Q value. 1.0 to 10.0. Default = 1.0.
Remarks

Deprecated and will be removed in a future release, to emulate with
**FMOD_DSP_TYPE_MULTIBAND_EQ**:

```cpp
// Configure a single band (band A) as a lowpass (all other bands default to off).
// 24dB rolloff to approximate the old effect curve.
// Cutoff frequency can be used the same as with the old effect.
// Resonance can be applied by setting the 'Q' value of the new effect.
FMOD_DSP_SetParameterInt(multiband, FMOD_DSP_MULTIBAND_EQ_A_FILTER,
FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_FREQUENCY,
FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_Q, res
```
See Also

- **DSP::setParameterFloat**
- **DSP::getParameterFloat**
- **FMOD_DSP_TYPE**
Firelight Technologies FMOD Studio API
FMOD_DSP_LOWPASS_SIMPLE

Parameter types for the FMOD_DSP_TYPE_LOWPASS_SIMPLE filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_LOWPASS_SIMPLE_CUTOFF
} FMOD_DSP_LOWPASS_SIMPLE;

JavaScript Syntax

FMOD.DSP_LOWPASS_SIMPLE_CUTOFF
Values

FMOD_DSP_LOWPASS_SIMPLE_CUTOFF

(Type: float) - Lowpass cutoff frequency in hz. 10.0 to 22000.0. Default = 5000.0
Remarks

Deprecated and will be removed in a future release, to emulate with `FMOD_DSP_TYPE_MULTIBAND_EQ`:

```c
// Configure a single band (band A) as a lowpass (all other bands default 12dB rolloff to approximate the old effect curve.
// Cutoff frequency can be used the same as with the old effect.
// Resonance / 'Q' should remain at default 0.707.
FMOD_DSP_SetParameterInt(multiband, FMOD_DSP_MULTIBAND_EQ_A_FILTER);
FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_FREQUENCY);
```

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.
See Also

- `DSP::setParameterFloat`
- `DSP::getParameterFloat`
- `FMOD_DSP_TYPE`
Firelight Technologies FMOD Studio API
Parameter types for the `FMOD_DSP_TYPE_MULTIBAND_EQ` filter.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_MULTIBAND_EQ_A_FILTER,
    FMOD_DSP_MULTIBAND_EQ_A_FREQUENCY,
    FMOD_DSP_MULTIBAND_EQ_A_Q,
    FMOD_DSP_MULTIBAND_EQ_A_GAIN,
    FMOD_DSP_MULTIBAND_EQ_B_FILTER,
    FMOD_DSP_MULTIBAND_EQ_B_FREQUENCY,
    FMOD_DSP_MULTIBAND_EQ_B_Q,
    FMOD_DSP_MULTIBAND_EQ_B_GAIN,
    FMOD_DSP_MULTIBAND_EQ_C_FILTER,
    FMOD_DSP_MULTIBAND_EQ_C_FREQUENCY,
    FMOD_DSP_MULTIBAND_EQ_C_Q,
    FMOD_DSP_MULTIBAND_EQ_C_GAIN,
    FMOD_DSP_MULTIBAND_EQ_D_FILTER,
    FMOD_DSP_MULTIBAND_EQ_D_FREQUENCY,
    FMOD_DSP_MULTIBAND_EQ_D_Q,
    FMOD_DSP_MULTIBAND_EQ_D_GAIN,
    FMOD_DSP_MULTIBAND_EQ_E_FILTER,
    FMOD_DSP_MULTIBAND_EQ_E_FREQUENCY,
    FMOD_DSP_MULTIBAND_EQ_E_Q,
    FMOD_DSP_MULTIBAND_EQ_E_GAIN
} FMOD_DSP_MULTIBAND_EQ;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_MULTIBAND_EQ_A_FILTER
FMOD.DSP_MULTIBAND_EQ_A_FREQUENCY
FMOD.DSP_MULTIBAND_EQ_A_Q
FMOD.DSP_MULTIBAND_EQ_A_GAIN
FMOD.DSP_MULTIBAND_EQ_B_FILTER
FMOD.DSP_MULTIBAND_EQ_B_FREQUENCY
FMOD.DSP_MULTIBAND_EQ_B_Q
FMOD.DSP_MULTIBAND_EQ_B_GAIN
FMOD.DSP_MULTIBAND_EQ_C_FILTER
FMOD.DSP_MULTIBAND_EQ_C_FREQUENCY
FMOD.DSP_MULTIBAND_EQ_C_Q
FMOD.DSP_MULTIBAND_EQ_C_GAIN
FMOD.DSP_MULTIBAND_EQ_D_FILTER
```
FMOD.DSP_MULTIBAND_EQ_D_FREQUENCY
FMOD.DSP_MULTIBAND_EQ_D_Q
FMOD.DSP_MULTIBAND_EQ_D_GAIN
FMOD.DSP_MULTIBAND_EQ_E_FILTER
FMOD.DSP_MULTIBAND_EQ_E_FREQUENCY
FMOD.DSP_MULTIBAND_EQ_E_Q
FMOD.DSP_MULTIBAND_EQ_E_GAIN
Values

*FMOD_DSP_MULTIBAND_EQ_A_FILTER*

(Type:int) - Band A: *FMOD_DSP_MULTIBAND_EQ_FILTER_TYPE* used to interpret the behavior of the remaining parameters. Default = *FMOD_DSP_MULTIBAND_EQ_FILTER_LOWPASS_12DB*

*FMOD_DSP_MULTIBAND_EQ_A_FREQUENCY*

(Type:float) - Band A: Significant frequency in Hz, cutoff [low/high pass, low/high shelf], center [notch, peaking, band-pass], phase transition point [all-pass]. 20 to 22000. Default = 8000.

*FMOD_DSP_MULTIBAND_EQ_A_Q*

(Type:float) - Band A: Quality factor, resonance [low/high pass], bandwidth [notch, peaking, band-pass], phase transition sharpness [all-pass], unused [low/high shelf]. 0.1 to 10.0. Default = 0.707.

*FMOD_DSP_MULTIBAND_EQ_A_GAIN*

(Type:float) - Band A: Boost or attenuation in dB [peaking, high/low shelf only]. -30 to 30. Default = 0.

*FMOD_DSP_MULTIBAND_EQ_B_FILTER*

(Type:int) - Band B: See Band A. Default = *FMOD_DSP_MULTIBAND_EQ_FILTER_DISABLED*

*FMOD_DSP_MULTIBAND_EQ_B_FREQUENCY*

(Type:float) - Band B: See Band A

*FMOD_DSP_MULTIBAND_EQ_B_Q*

(Type:float) - Band B: See Band A

*FMOD_DSP_MULTIBAND_EQ_B_GAIN*
(Type:float) - Band B: See Band A

_FMOD_DSP_MULTIBAND_EQ_C_FILTER_

(Type:int) - Band C: See Band A. Default =
_FMOD_DSP_MULTIBAND_EQ_FILTER_DISABLED_

_FMOD_DSP_MULTIBAND_EQ_C_FREQUENCY_

(Type:float) - Band C: See Band A.

_FMOD_DSP_MULTIBAND_EQ_C_Q_

(Type:float) - Band C: See Band A.

_FMOD_DSP_MULTIBAND_EQ_C_GAIN_

(Type:float) - Band C: See Band A.

_FMOD_DSP_MULTIBAND_EQ_D_FILTER_

(Type:int) - Band D: See Band A. Default =
_FMOD_DSP_MULTIBAND_EQ_FILTER_DISABLED_

_FMOD_DSP_MULTIBAND_EQ_D_FREQUENCY_

(Type:float) - Band D: See Band A.

_FMOD_DSP_MULTIBAND_EQ_D_Q_

(Type:float) - Band D: See Band A.

_FMOD_DSP_MULTIBAND_EQ_D_GAIN_

(Type:float) - Band D: See Band A.

_FMOD_DSP_MULTIBAND_EQ_E_FILTER_

(Type:int) - Band E: See Band A. Default =
_FMOD_DSP_MULTIBAND_EQ_FILTER_DISABLED_
**FMOD_DSP_MULTIBAND_EQ_E_FREQUENCY**

(Type: float) - Band E: See Band A.

**FMOD_DSP_MULTIBAND_EQ_E_Q**

(Type: float) - Band E: See Band A.

**FMOD_DSP_MULTIBAND_EQ_E_GAIN**

(Type: float) - Band E: See Band A.
Remarks

Flexible five band parametric equalizer.
See Also

- DSP::setParameterInt
- DSP::getParameterInt
- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
Filter types for FMOD_DSP_MULTIBAND_EQ.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_MULTIBAND_EQ_FILTER_DISABLED,
    FMOD_DSP_MULTIBAND_EQ_FILTER_LOWPASS_12DB,
    FMOD_DSP_MULTIBAND_EQ_FILTER_LOWPASS_24DB,
    FMOD_DSP_MULTIBAND_EQ_FILTER_LOWPASS_48DB,
    FMOD_DSP_MULTIBAND_EQ_FILTER_HIGHPASS_12DB,
    FMOD_DSP_MULTIBAND_EQ_FILTER_HIGHPASS_24DB,
    FMOD_DSP_MULTIBAND_EQ_FILTER_HIGHPASS_48DB,
    FMOD_DSP_MULTIBAND_EQ_FILTER_LOWSHELF,
    FMOD_DSP_MULTIBAND_EQ_FILTER_HIGHSHELF,
    FMOD_DSP_MULTIBAND_EQ_FILTER_PEAKING,
    FMOD_DSP_MULTIBAND_EQ_FILTER_BANDPASS,
    FMOD_DSP_MULTIBAND_EQ_FILTER_NOTCH,
    FMOD_DSP_MULTIBAND_EQ_FILTER_ALLPASS
} FMOD_DSP_MULTIBAND_EQ_FILTER_TYPE;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_MULTIBAND_EQ_FILTER_DISABLED
FMOD.DSP_MULTIBAND_EQ_FILTER_LOWPASS_12DB
FMOD.DSP_MULTIBAND_EQ_FILTER_LOWPASS_24DB
FMOD.DSP_MULTIBAND_EQ_FILTER_LOWPASS_48DB
FMOD.DSP_MULTIBAND_EQ_FILTER_HIGHPASS_12DB
FMOD.DSP_MULTIBAND_EQ_FILTER_HIGHPASS_24DB
FMOD.DSP_MULTIBAND_EQ_FILTER_HIGHPASS_48DB
FMOD.DSP_MULTIBAND_EQ_FILTER_LOWSHELF
FMOD.DSP_MULTIBAND_EQ_FILTER_HIGHSHELF
FMOD.DSP_MULTIBAND_EQ_FILTER_PEAKING
FMOD.DSP_MULTIBAND_EQ_FILTER_BANDPASS
FMOD.DSP_MULTIBAND_EQ_FILTER_NOTCH
FMOD.DSP_MULTIBAND_EQ_FILTER_ALLPASS
```
Values

* FMOD_DSP_MULTIBAND_EQ_FILTER_DISABLED
  Disabled filter, no processing.

* FMOD_DSP_MULTIBAND_EQ_FILTER_LOWPASS_12DB
  Resonant low-pass filter, attenuates frequencies (12dB per octave) above a given point (with specified resonance) while allowing the rest to pass.

* FMOD_DSP_MULTIBAND_EQ_FILTER_LOWPASS_24DB
  Resonant low-pass filter, attenuates frequencies (24dB per octave) above a given point (with specified resonance) while allowing the rest to pass.

* FMOD_DSP_MULTIBAND_EQ_FILTER_LOWPASS_48DB
  Resonant low-pass filter, attenuates frequencies (48dB per octave) above a given point (with specified resonance) while allowing the rest to pass.

* FMOD_DSP_MULTIBAND_EQ_FILTER_HIGHPASS_12DB
  Resonant low-pass filter, attenuates frequencies (12dB per octave) below a given point (with specified resonance) while allowing the rest to pass.

* FMOD_DSP_MULTIBAND_EQ_FILTER_HIGHPASS_24DB
  Resonant low-pass filter, attenuates frequencies (24dB per octave) below a given point (with specified resonance) while allowing the rest to pass.

* FMOD_DSP_MULTIBAND_EQ_FILTER_HIGHPASS_48DB
  Resonant low-pass filter, attenuates frequencies (48dB per octave) below a given point (with specified resonance) while allowing the rest to pass.

* FMOD_DSP_MULTIBAND_EQ_FILTER_LOWSHELF
  Low-shelf filter, boosts or attenuates frequencies (with specified gain) below a
given point while allowing the rest to pass.

*FMOD_DSP_MULTIBAND_EQ_FILTER_HIGHSHELF*

High-shelf filter, boosts or attenuates frequencies (with specified gain) above a given point while allowing the rest to pass.

*FMOD_DSP_MULTIBAND_EQ_FILTER_PEAKING*

Peaking filter, boosts or attenuates frequencies (with specified gain) at a given point (with specified bandwidth) while allowing the rest to pass.

*FMOD_DSP_MULTIBAND_EQ_FILTER_BANDPASS*

Band-pass filter, allows frequencies at a given point (with specified bandwidth) to pass while attenuating frequencies outside this range.

*FMOD_DSP_MULTIBAND_EQ_FILTER_NOTCH*

Notch or band-reject filter, attenuates frequencies at a given point (with specified bandwidth) while allowing frequencies outside this range to pass.

*FMOD_DSP_MULTIBAND_EQ_FILTER_ALLPASS*

All-pass filter, allows all frequencies to pass, but changes the phase response at a given point (with specified sharpness).
See Also

- **FMOD_DSP_MULTIBAND_EQ**
Firelight Technologies FMOD Studio API
FMOD_DSP_NORMALIZE

Parameter types for the FMOD_DSP_TYPE_NORMALIZE filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_NORMALIZE_FADETIME,
    FMOD_DSP_NORMALIZE_THRESHHOLD,
    FMOD_DSP_NORMALIZE_MAXAMP
} FMOD_DSP_NORMALIZE;

JavaScript Syntax

FMOD.DSP_NORMALIZE_FADETIME
FMOD.DSP_NORMALIZE_THRESHHOLD
FMOD.DSP_NORMALIZE_MAXAMP
Values

\textit{FMOD\_DSP\_NORMALIZE\_FADETIME}

(Type: float) - Time to ramp the silence to full in ms. 0.0 to 20000.0. Default = 5000.0.

\textit{FMOD\_DSP\_NORMALIZE\_THRESHHOLD}

(Type: float) - Lower volume range threshold to ignore. 0.0 to 1.0. Default = 0.1. Raise higher to stop amplification of very quiet signals.

\textit{FMOD\_DSP\_NORMALIZE\_MAXAMP}

(Type: float) - 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::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_OBJECTPAN

Parameter types for the FMOD_DSP_TYPE_OBJECTPAN DSP. 3D Object panners are meant for hardware 3d object systems like Dolby Atmos or Sony Morpheus. These object panners take input in, and send it to the 7.1 bed, but do not send the signal further down the DSP chain (the output of the dsp is silence).

C/C++ Syntax

typedef enum {
    FMOD_DSP_OBJECTPAN_3D_POSITION,
    FMOD_DSP_OBJECTPAN_3D_ROLLOFF,
    FMOD_DSP_OBJECTPAN_3D_MIN_DISTANCE,
    FMOD_DSP_OBJECTPAN_3D_MAX_DISTANCE,
    FMOD_DSP_OBJECTPAN_3D_EXTENT_MODE,
    FMOD_DSP_OBJECTPAN_3D_SOUND_SIZE,
    FMOD_DSP_OBJECTPAN_3D_MIN_EXTENT,
    FMOD_DSP_OBJECTPAN_OVERALL_GAIN,
    FMOD_DSP_OBJECTPAN_OUTPUTGAIN
} FMOD_DSP_OBJECTPAN;

JavaScript Syntax

FMOD_DSP_OBJECTPAN_3D_POSITION
FMOD_DSP_OBJECTPAN_3D_ROLLOFF
FMOD_DSP_OBJECTPAN_3D_MIN_DISTANCE
FMOD_DSP_OBJECTPAN_3D_MAX_DISTANCE
FMOD_DSP_OBJECTPAN_3D_EXTENT_MODE
FMOD_DSP_OBJECTPAN_3D_SOUND_SIZE
FMOD_DSP_OBJECTPAN_3D_MIN_EXTENT
FMOD_DSP_OBJECTPAN_OVERALL_GAIN
FMOD_DSP_OBJECTPAN_OUTPUTGAIN
Values

\textbf{FMOD\_DSP\_OBJECTPAN\_3D\_POSITION}

(Type: data) - 3D Position. data of type \textbf{FMOD\_DSP\_PARAMETER\_3D\_ATTRIBUTES\_MULTI}

\textbf{FMOD\_DSP\_OBJECTPAN\_3D\_ROLLOFF}

(Type: int) - 3D Rolloff. \textbf{FMOD\_DSP\_PAN\_3D\_ROLLOFF\_LINEARSQUARED} to \textbf{FMOD\_DSP\_PAN\_3D\_ROLLOFF\_CUSTOM}. Default = \textbf{FMOD\_DSP\_PAN\_3D\_ROLLOFF\_LINEARSQUARED}.

\textbf{FMOD\_DSP\_OBJECTPAN\_3D\_MIN\_DISTANCE}

(Type: float) - 3D Min Distance. 0.0 to 1e+18f. Default = 1.0.

\textbf{FMOD\_DSP\_OBJECTPAN\_3D\_MAX\_DISTANCE}

(Type: float) - 3D Max Distance. 0.0 to 1e+18f. Default = 20.0.

\textbf{FMOD\_DSP\_OBJECTPAN\_3D\_EXTENT\_MODE}

(Type: int) - 3D Extent Mode. \textbf{FMOD\_DSP\_PAN\_3D\_EXTENT\_MODE\_AUTO} to \textbf{FMOD\_DSP\_PAN\_3D\_EXTENT\_MODE\_OFF}. Default = \textbf{FMOD\_DSP\_PAN\_3D\_EXTENT\_MODE\_AUTO}.

\textbf{FMOD\_DSP\_OBJECTPAN\_3D\_SOUND\_SIZE}

(Type: float) - 3D Sound Size. 0.0 to 1e+18f. Default = 0.0.

\textbf{FMOD\_DSP\_OBJECTPAN\_3D\_MIN\_EXTENT}

(Type: float) - 3D Min Extent. 0.0 (degrees) to 360.0 (degrees). Default = 0.0.

\textbf{FMOD\_DSP\_OBJECTPAN\_OVERALL\_GAIN}

(Type: data) - Overall gain. For information only, not set by user. Data of type
**FMOD_DSP_PARAMETER_DATA_TYPE_OVERALLGAIN** to provide to FMOD, to allow FMOD to know the DSP is scaling the signal for virtualization purposes.

**FMOD_DSP_OBJECTPAN_OUTPUTGAIN**

(Type: float) - Output gain level. 0.0 to 1.0 linear scale. For the user to scale the output of the object panner's signal.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterInt
- DSP::getParameterInt
- DSP::setParameterData
- DSP::getParameterData
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
**FMOD_DSP_OSCILLATOR**

Parameter types for the `FMOD_DSP_TYPE_OSCILLATOR` filter.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_OSCILLATOR_TYPE,
    FMOD_DSP_OSCILLATOR_RATE
} FMOD_DSP_OSCILLATOR;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_OSCILLATOR_TYPE
FMOD.DSP_OSCILLATOR_RATE
```
Values

*FMOD_DSP_OSCILLATOR_TYPE*

(Type: int) - Waveform type. 0 = sine. 1 = square. 2 = sawup. 3 = sawdown. 4 = triangle. 5 = noise.

*FMOD_DSP_OSCILLATOR_RATE*

(Type: float) - Frequency of the sinewave in hz. 1.0 to 22000.0. Default = 220.0.
See Also

- DSP::setParameterFloat
- DSP::setParameterInt
- DSP::getParameterFloat
- DSP::getParameterInt
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_PAN

Parameter types for the FMOD_DSP_TYPE_PAN DSP.

C/C++ Syntax

typedef enum {
    FMOD_DSP_PAN_MODE,
    FMOD_DSP_PAN_2D_STEREO_POSITION,
    FMOD_DSP_PAN_2D_DIRECTION,
    FMOD_DSP_PAN_2D_EXTENT,
    FMOD_DSP_PAN_2D_ROTATION,
    FMOD_DSP_PAN_2D_LFE_LEVEL,
    FMOD_DSP_PAN_2D_STEREO_MODE,
    FMOD_DSP_PAN_2D_STEREO_SEPARATION,
    FMOD_DSP_PAN_2D_STEREO_AXIS,
    FMOD_DSP_PAN_ENABLED_SPEAKERS,
    FMOD_DSP_PAN_3D_POSITION,
    FMOD_DSP_PAN_3D_ROLLOFF,
    FMOD_DSP_PAN_3D_MIN_DISTANCE,
    FMOD_DSP_PAN_3D_MAX_DISTANCE,
    FMOD_DSP_PAN_3D_EXTENT_MODE,
    FMOD_DSP_PAN_3D_SOUND_SIZE,
    FMOD_DSP_PAN_3D_MIN_EXTENT,
    FMOD_DSP_PAN_3D_PAN_BLEND,
    FMOD_DSP_PAN_LFE_UPMIX_ENABLED,
    FMOD_DSP_PAN_OVERALL_GAIN,
    FMOD_DSP_PAN_SURROUND_SPEAKER_MODE,
    FMOD_DSP_PAN_2D_HEIGHT_BLEND
} FMOD_DSP_PAN;

JavaScript Syntax

FMOD.DSP_PAN_MODE
FMOD.DSP_PAN_2D_STEREO_POSITION
FMOD.DSP_PAN_2D_DIRECTION
FMOD.DSP_PAN_2D_EXTENT
FMOD.DSP_PAN_2D_ROTATION
FMOD.DSP_PAN_2D_LFE_LEVEL
FMOD.DSP_PAN_2D_STEREO_MODE
FMOD.DSP_PAN_2D_STEREO_SEPARATION
FMOD.DSP_PAN_2D_STEREO_AXIS
FMOD.DSP_PAN_ENABLED_SPEAKERS
FMOD.DSP_PAN_3D_POSITION
FMOD.DSP_PAN_3D_ROLLOFF
FMOD.DSP_PAN_3D_MIN_DISTANCE
FMOD.DSP_PAN_3D_MAX_DISTANCE
FMOD.DSP_PAN_3D_EXTENT_MODE
FMOD.DSP_PAN_3D_SOUND_SIZE
FMOD.DSP_PAN_3D_MIN_EXTENT
FMOD.DSP_PAN_3D_PAN_BLEND
FMOD.DSP_PAN_LFE_UPMIX_ENABLED
FMOD.DSP_PAN_OVERALL_GAIN
FMOD.DSP_PAN_SURROUND_SPEAKER_MODE
Values

**FMOD_DSP_PAN_MODE**

(Type:int) - Panner mode. **FMOD_DSP_PAN_MODE_MONO** for mono down-mix, **FMOD_DSP_PAN_MODE_STEREO** for stereo panning or **FMOD_DSP_PAN_MODE_SURROUND** for surround panning. Default = **FMOD_DSP_PAN_MODE_SURROUND**

**FMOD_DSP_PAN_2D_STEREO_POSITION**

(Type:float) - 2D Stereo pan position. -100.0 to 100.0. Default = 0.0.

**FMOD_DSP_PAN_2D_DIRECTION**

(Type:float) - 2D Surround pan direction. Direction from center point of panning circle. -180.0 (degrees) to 180.0 (degrees). 0 = front center, -180 or +180 = rear speakers center point. Default = 0.0.

**FMOD_DSP_PAN_2D_EXTENT**

(Type:float) - 2D Surround pan extent. Distance from center point of panning circle. 0.0 (degrees) to 360.0 (degrees). Default = 360.0.

**FMOD_DSP_PAN_2D_ROTATION**

(Type:float) - 2D Surround pan rotation. -180.0 (degrees) to 180.0 (degrees). Default = 0.0.

**FMOD_DSP_PAN_2D_LFE_LEVEL**

(Type:float) - 2D Surround pan LFE level. 2D LFE level in dB. -80.0 (db) to 20.0 (db). Default = 0.0.

**FMOD_DSP_PAN_2D_STEREO_MODE**

(Type:int) - Stereo-To-Surround Mode. **FMOD_DSP_PAN_2D_STEREO_MODE_DISTRIBUTED** to **FMOD_DSP_PAN_2D_STEREO_MODE_DISCRETE**. Default =
**FMOD_DSP_PAN_2D_STEREO_MODE_DISCRETE.**

**FMOD_DSP_PAN_2D_STEREO_SEPARATION**

(Type:float) - Stereo-To-Surround Stereo For **FMOD_DSP_PAN_2D_STEREO_MODE_DISCRETE** mode. Separation/width of L/R parts of stereo sound. -180.0 (degrees) to +180.0 (degrees). Default = 60.0.

**FMOD_DSP_PAN_2D_STEREO_AXIS**

(Type:float) - Stereo-To-Surround Stereo For **FMOD_DSP_PAN_2D_STEREO_MODE_DISCRETE** mode. Axis/rotation of L/R parts of stereo sound. -180.0 (degrees) to +180.0 (degrees). Default = 0.0.

**FMOD_DSP_PAN_ENABLED_SPEAKERS**

(Type:int) - Speakers Enabled. Bitmask for each speaker from 0 to 32 to be considered by panner. Use to disable speakers from being panned to. 0 to 0xFFF. Default = 0xFFF (All on).

**FMOD_DSP_PAN_3D_POSITION**

(Type:data) - 3D Position. Data of type **FMOD_DSP_PARAMETER_3DATTRIBUTES_MULTI**. See remarks on what to fill out.

**FMOD_DSP_PAN_3D_ROLLOFF**

(Type:int) - 3D Rolloff. **FMOD_DSP_PAN_3D_ROLLOFF_LINEARSQUARED** to **FMOD_DSP_PAN_3D_ROLLOFF_CUSTOM**. Default = **FMOD_DSP_PAN_3D_ROLLOFF_LINEARSQUARED**.

**FMOD_DSP_PAN_3D_MIN_DISTANCE**

(Type:float) - 3D Min Distance. 0.0 to 1e+18f. Default = 1.0.

**FMOD_DSP_PAN_3D_MAX_DISTANCE**
(Type:float) - 3D Max Distance. 0.0 to 1e+18f. Default = 20.0.

**FMOD_DSP_PAN_3D_EXTENT_MODE**

(Type:int) - 3D Extent Mode. **FMOD_DSP_PAN_3D_EXTENT_MODE_AUTO** to **FMOD_DSP_PAN_3D_EXTENT_MODE_OFF**. Default = **FMOD_DSP_PAN_3D_EXTENT_MODE_AUTO**.

**FMOD_DSP_PAN_3D_SOUND_SIZE**

(Type:float) - 3D Sound Size. 0.0 to 1e+18f. Default = 0.0.

**FMOD_DSP_PAN_3D_MIN_EXTENT**

(Type:float) - 3D Min Extent. 0.0 (degrees) to 360.0 (degrees). Default = 0.0.

**FMOD_DSP_PAN_3D_PAN_BLEND**

(Type:float) - 3D Pan Blend. 0.0 (fully 2D) to 1.0 (fully 3D). Default = 0.0.

**FMOD_DSP_PAN_LFE_UPMIX_ENABLED**

(Type:int) - LFE Upmix Enabled. Determines whether non-LFE source channels should mix to the LFE or leave it alone. 0 (off) to 1 (on). Default = 0 (off).

**FMOD_DSP_PAN_OVERALL_GAIN**

(Type:data) - Overall gain. For information only, not set by user. Data of type **FMOD_DSP_PARAMETER_DATA_TYPE_OVERALLGAIN** to provide to FMOD, to allow FMOD to know the DSP is scaling the signal for virtualization purposes.

**FMOD_DSP_PAN_SURROUND_SPEAKER_MODE**

(Type:int) - Surround speaker mode. Target speaker mode for surround panning. Default = **FMOD_SPEAKERMODE_DEFAULT**.

**FMOD_DSP_PAN_2D_HEIGHT_BLEND**

(Type:float) - 2D Height blend. When the input or **FMOD_DSP_PAN_SURROUND_SPEAKER_MODE** has height speakers,
control the blend between ground and height. -1.0 (push top speakers to ground), 0.0 (preserve top / ground separation), 1.0 (push ground speakers to top). Default = 0.0.
Remarks

**FMOD_DSP_PAN_3D_PAN_BLEND** controls the percentage of the effect supplied by **FMOD_DSP_PAN_2D_DIRECTION** and **FMOD_DSP_PAN_2D_EXTENT**.

For **FMOD_DSP_PAN_3D_POSITION**, the following members in the **FMOD_DSP_PARAMETER_3DATTRIBUTES_MULTI** struct should be non-zero. - numlisteners - This is typically 1, can be up to 8. Typically more than 1 is only used for split screen purposes. The FMOD Panner will average angles and produce the best compromise for panning and attenuation. - relative[listenernum].position - This is the delta between the listener position and the sound position. Typically the listener position is subtracted from the sound position. - relative[listenernum].forward - This is the sound's forward vector. Optional, set to 0,0,1 if not needed. This is only relevant for more than mono sounds in 3D, that are spread amongst the destination speakers at the time of panning. If the sound rotates then the L/R part of a stereo sound will rotate amongst its destination speakers. If the sound has moved and pinpointed into a single speaker, rotation of the sound will have no effect as at that point the channels are collapsed into a single point.

For **FMOD_DSP_PAN_2D_STEREO_MODE**, when it is set to **FMOD_DSP_PAN_2D_STEREO_MODE_DISCRETE**, only **FMOD_DSP_PAN_2D_STEREO_SEPARATION** and **FMOD_DSP_PAN_2D_STEREO_AXIS** are used. When it is set to **FMOD_DSP_PAN_2D_STEREO_MODE_DISTRIBUTED**, then standard **FMOD_DSP_PAN_2D_DIRECTION/FMOD_DSP_PAN_2D_EXTENT** parameters are used.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterInt
- DSP::getParameterInt
- DSP::setParameterData
- DSP::getParameterData
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_PAN_2D_STEREO_MODE

Parameter values for the FMOD_DSP_PAN_2D_STEREO_MODE parameter of the FMOD_DSP_TYPE_PAN DSP.

C/C++ Syntax

typedef enum {
    FMOD_DSP_PAN_2D_STEREO_MODE_DISTRIBUTED,
    FMOD_DSP_PAN_2D_STEREO_MODE_DISCRETE
} FMOD_DSP_PAN_2D_STEREO_MODE_TYPE;

JavaScript Syntax

FMOD.DSP_PAN_2D_STEREO_MODE_DISTRIBUTED
FMOD.DSP_PAN_2D_STEREO_MODE_DISCRETE
Values

**FMOD_DSP_PAN_2D_STEREO_MODE_DISTRIBUTED**

The parts of a stereo sound are spread around destination speakers based on
**FMOD_DSP_PAN_2D_EXTENT / FMOD_DSP_PAN_2D_DIRECTION**

**FMOD_DSP_PAN_2D_STEREO_MODE_DISCRETE**

The L/R parts of a stereo sound are rotated around a circle based on
**FMOD_DSP_PAN_2D_STEREO_AXIS / FMOD_DSP_PAN_2D_STEREO_SEPARATION**.
See Also

- **FMOD_DSP_PAN**
Firelight Technologies FMOD Studio API
FMOD_DSP_PAN_3D_EXTENT_MODE

Parameter values for the FMOD_DSP_PAN_3D_EXTENT_MODE parameter of the FMOD_DSP_TYPE_PAN DSP.

**C/C++ Syntax**

typedef enum {
    FMOD_DSP_PAN_3D_EXTENT_MODE_AUTO,
    FMOD_DSP_PAN_3D_EXTENT_MODE_USER,
    FMOD_DSP_PAN_3D_EXTENT_MODE_OFF
} FMOD_DSP_PAN_3D_EXTENT_MODE_TYPE;

**JavaScript Syntax**

FMOD.DSP_PAN_3D_EXTENT_MODE_AUTO
FMOD.DSP_PAN_3D_EXTENT_MODE_USER
FMOD.DSP_PAN_3D_EXTENT_MODE_OFF
Values

FMOD_DSP_PAN_3D_EXTENT_MODE_AUTO
FMOD_DSP_PAN_3D_EXTENT_MODE_USER
FMOD_DSP_PAN_3D_EXTENT_MODE_OFF
See Also

- FMOD_DSP_PAN
Firelight Technologies FMOD Studio API
Parameter values for the FMOD_DSP_PAN_3D_ROLLOFF parameter of the FMOD_DSP_TYPE_PAN DSP.

C/C++ Syntax

typedef enum {
    FMOD_DSP_PAN_3D_ROLLOFF_LINEARSQUARED,
    FMOD_DSP_PAN_3D_ROLLOFF_LINEAR,
    FMOD_DSP_PAN_3D_ROLLOFF_INVERSE,
    FMOD_DSP_PAN_3D_ROLLOFF_INVERSETAPERED,
    FMOD_DSP_PAN_3D_ROLLOFF_CUSTOM
} FMOD_DSP_PAN_3D_ROLLOFF_TYPE;

JavaScript Syntax

FMOD_DSP_PAN_3D_ROLLOFF_LINEARSQUARED
FMOD_DSP_PAN_3D_ROLLOFF_LINEAR
FMOD_DSP_PAN_3D_ROLLOFF_INVERSE
FMOD_DSP_PAN_3D_ROLLOFF_INVERSETAPERED
FMOD_DSP_PAN_3D_ROLLOFF_CUSTOM
Values

FMOD_DSP_PAN_3D_ROLLOFF_LINEARSQUARED
FMOD_DSP_PAN_3D_ROLLOFF_LINEAR
FMOD_DSP_PAN_3D_ROLLOFF_INVERSE
FMOD_DSP_PAN_3D_ROLLOFF_INVERSETAPERED
FMOD_DSP_PAN_3D_ROLLOFF_CUSTOM
See Also

- **FMOD_DSP_PAN**
Firelight Technologies FMOD Studio API
FMOD_DSP_PAN_MODE_TYPE

Parameter values for the FMOD_DSP_PAN_MODE parameter of the FMOD_DSP_TYPE_PAN DSP.

C/C++ Syntax

typedef enum {
    FMOD_DSP_PAN_MODE_MONO,
    FMOD_DSP_PAN_MODE_STEREO,
    FMOD_DSP_PAN_MODE_SURROUND
} FMOD_DSP_PAN_MODE_TYPE;

JavaScript Syntax

FMOD.DSP_PAN_MODE_MONO
FMOD.DSP_PAN_MODE_STEREO
FMOD.DSP_PAN_MODE_SURROUND
Values

`FMOD_DSP_PAN_MODE_MONO`

`FMOD_DSP_PAN_MODE_STEREO`

`FMOD_DSP_PAN_MODE_SURROUND`
See Also

- FMOD_DSP_PAN
Firelight Technologies FMOD Studio API
**FMOD_DSP_PAN_SURROUND_FLAGS**

Flags for the FMOD_DSP_PAN_SUMSURROUNDMATRIX_FUNC function.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_PAN_SURROUND_DEFAULT,
    FMOD_DSP_PAN_SURROUND_ROTATION_NOT_BIASED
} FMOD_DSP_PAN_SURROUND_FLAGS;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_PAN_SURROUND_DEFAULT
FMOD.DSP_PAN_SURROUND_ROTATION_NOT_BIASED
FMOD.DSP_PAN_SURROUND_FLAGS_FORCEINT
```
Values

* FMOD_DSP_PAN_SURROUND_DEFAULT
* FMOD_DSP_PAN_SURROUND_ROTATION_NOT_BIASED
Remarks

This functionality is experimental, please contact support@fmod.org for more information.
See Also

- `FMOD_DSP_STATE_PAN_FUNCTIONS`
Firelight Technologies FMOD Studio API
**FMOD_DSP_PARAMEQ**

Parameter types for the `FMOD_DSP_TYPE_PARAMEQ` filter.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_PARAMEQ_CENTER,
    FMOD_DSP_PARAMEQ_BANDWIDTH,
    FMOD_DSP_PARAMEQ_GAIN
} FMOD_DSP_PARAMEQ;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_PARAMEQ_CENTER
FMOD.DSP_PARAMEQ_BANDWIDTH
FMOD.DSP_PARAMEQ_GAIN
```
Values

*FMOD_DSP_PARAMEQ_CENTER*

(Type: float) - Frequency center. 20.0 to 22000.0. Default = 8000.0.

*FMOD_DSP_PARAMEQ_BANDWIDTH*

(Type: float) - Octave range around the center frequency to filter. 0.2 to 5.0. Default = 1.0.

*FMOD_DSP_PARAMEQ_GAIN*

(Type: float) - Frequency Gain in dB. -30 to 30. Default = 0.
Remarks

Deprecated and will be removed in a future release, to emulate with `FMOD_DSP_TYPE_MULTIBAND_EQ`:

```cpp
// Configure a single band (band A) as a peaking EQ (all other bands default to off).
// Center frequency can be used as with the old effect.
// Bandwidth can be applied by setting the 'Q' value of the new effect.
// Gain at the center frequency can be used the same as with the old
FMOD_DSP_SetParameterInt(multiband, FMOD_DSP_MULTIBAND_EQ_A_FILTER,
FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_FREQUENCY
FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_Q, bandwidth);
FMOD_DSP_SetParameterFloat(multiband, FMOD_DSP_MULTIBAND_EQ_A_GAIN,
```

Parametric EQ is a single band peaking EQ filter that attenuates or amplifies a selected frequency and its neighbouring frequencies.

When a frequency has its gain set to 1.0, the sound will be unaffected and represents the original signal exactly.
See Also

- **DSP::setParameterFloat**
- **DSP::getParameterFloat**
- **FMOD_DSP_TYPE**
Firelight Technologies FMOD Studio API
Built-in types for the 'datatype' member of `FMOD_DSP_PARAMETER_DESC_DATA`. Data parameters of type other than `FMOD_DSP_PARAMETER_DATA_TYPE_USER` will be treated specially by the system.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_PARAMETER_DATA_TYPE_USER,
    FMOD_DSP_PARAMETER_DATA_TYPE_OVERALLGAIN,
    FMOD_DSP_PARAMETER_DATA_TYPE_3DATTRIBUTES,
    FMOD_DSP_PARAMETER_DATA_TYPE_SIDECHAIN,
    FMOD_DSP_PARAMETER_DATA_TYPE_FFT,
    FMOD_DSP_PARAMETER_DATA_TYPE_3DATTRIBUTES_MULTI
} FMOD_DSP_PARAMETER_DATA_TYPE;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_PARAMETER_DATA_TYPE_USER
FMOD.DSP_PARAMETER_DATA_TYPE_OVERALLGAIN
FMOD.DSP_PARAMETER_DATA_TYPE_3DATTRIBUTES
FMOD.DSP_PARAMETER_DATA_TYPE_SIDECHAIN
FMOD.DSP_PARAMETER_DATA_TYPE_FFT
FMOD.DSP_PARAMETER_DATA_TYPE_3DATTRIBUTES_MULTI
```
Values

*FMOD_DSP_PARAMETER_DATA_TYPE_USER*

The default data type. All user data types should be 0 or above.

*FMOD_DSP_PARAMETER_DATA_TYPE_OVERALLGAIN*

The data type for *FMOD_DSP_PARAMETER_OVERALLGAIN* parameters. There should a maximum of one per DSP.

*FMOD_DSP_PARAMETER_DATA_TYPE_3DATTRIBUTES*

The data type for *FMOD_DSP_PARAMETER_3DATTRIBUTES* parameters. There should a maximum of one per DSP.

*FMOD_DSP_PARAMETER_DATA_TYPE_SIDECHAIN*

The data type for *FMOD_DSP_PARAMETER_SIDECHAIN* parameters. There should a maximum of one per DSP.

*FMOD_DSP_PARAMETER_DATA_TYPE_FFT*

The data type for *FMOD_DSP_PARAMETER_FFT* parameters. There should a maximum of one per DSP.

*FMOD_DSP_PARAMETER_DATA_TYPE_3DATTRIBUTES_MULTI*

The data type for *FMOD_DSP_PARAMETER_3DATTRIBUTES_MULTI* parameters. There should a maximum of one per DSP.
See Also

- **FMOD_DSP_PARAMETER_DESC_DATA**
- **FMOD_DSP_PARAMETER_OVERALLGAIN**
- **FMOD_DSP_PARAMETER_3DATRIBUTES**
- **FMOD_DSP_PARAMETER_3DATRIBUTES_MULTI**
- **FMOD_DSP_PARAMETER_SIDECHAIN**
- **DSP::getParameterData**
- **DSP::setParameterData**
Firelight Technologies FMOD Studio API
FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE

DSP float parameter mappings. These determine how values are mapped across dials and automation curves.

C/C++ Syntax

typedef enum {
    FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE_LINEAR,
    FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE_AUTO,
    FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE_PIECEWISE_LINEAR
} FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE;

JavaScript Syntax

FMOD.DSP_PARAMETER_FLOAT_MAPPING_TYPE_LINEAR
FMOD.DSP_PARAMETER_FLOAT_MAPPING_TYPE_AUTO
FMOD.DSP_PARAMETER_FLOAT_MAPPING_TYPE_PIECEWISE_LINEAR
FMOD.DSP_PARAMETER_FLOAT_MAPPING_TYPE_FORCEINT
**Values**

*FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE_LINEAR*

Values mapped linearly across range.

*FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE_AUTO*

A mapping is automatically chosen based on range and units. See remarks.

*FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE PIECEWISE_LINEAR*

Values mapped in a piecewise linear fashion defined by *FMOD_DSP_PARAMETER_FLOAT_MAPPING_PIECEWISE_LINEAR*. 
Remarks

**FMOD_DSP_PARAMETER_FLOAT_MAPPING_TYPE_AUTO** generates a mapping based on range and units. For example, if the units are in Hertz and the range is within the audio spectrum, a Bark scale will be chosen. Logarithmic scales may also be generated for ranges above zero spanning several orders of magnitude.
See Also

- FMOD_DSP_PARAMETER_FLOAT_MAPPING
Firelight Technologies FMOD Studio API
FMOD_DSP_PARAMETER_TYPE

DSP parameter types.

C/C++ Syntax

typedef enum {
    FMOD_DSP_PARAMETER_TYPE_FLOAT,
    FMOD_DSP_PARAMETER_TYPE_INT,
    FMOD_DSP_PARAMETER_TYPE_BOOL,
    FMOD_DSP_PARAMETER_TYPE_DATA,
    FMOD_DSP_PARAMETER_TYPE_MAX
} FMOD_DSP_PARAMETER_TYPE;

JavaScript Syntax

FMOD.DSP_PARAMETER_TYPE_FLOAT
FMOD.DSP_PARAMETER_TYPE_INT
FMOD.DSP_PARAMETER_TYPE_BOOL
FMOD.DSP_PARAMETER_TYPE_DATA
FMOD.DSP_PARAMETER_TYPE_MAX
FMOD.DSP_PARAMETER_TYPE_FORCEINT
Values

`FMOD_DSP_PARAMETER_TYPE_FLOAT`

`FMOD_DSP_PARAMETER_DESC` will use the `FMOD_DSP_PARAMETER_DESC_FLOAT`.

`FMOD_DSP_PARAMETER_TYPE_INT`

`FMOD_DSP_PARAMETER_DESC` will use the `FMOD_DSP_PARAMETER_DESC_INT`.

`FMOD_DSP_PARAMETER_TYPE_BOOL`

`FMOD_DSP_PARAMETER_DESC` will use the `FMOD_DSP_PARAMETER_DESC_BOOL`.

`FMOD_DSP_PARAMETER_TYPE_DATA`

`FMOD_DSP_PARAMETER_DESC` will use the `FMOD_DSP_PARAMETER_DESC_DATA`.

`FMOD_DSP_PARAMETER_TYPE_MAX`

Maximum number of DSP parameter types.
See Also

- FMOD_DSP_PARAMETER_DESC
Firelight Technologies FMOD Studio API
FMOD_DSP_PITCHSHIFT

Parameter types for the FMOD_DSP_TYPE_PITCHSHIFT filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_PITCHSHIFT_PITCH,
    FMOD_DSP_PITCHSHIFT_FFTSIZE,
    FMOD_DSP_PITCHSHIFT_OVERLAP,
    FMOD_DSP_PITCHSHIFT_MAXCHANNELS
} FMOD_DSP_PITCHSHIFT;

JavaScript Syntax

FMOD.DSP_PITCHSHIFT_PITCH
FMOD.DSP_PITCHSHIFT_FFTSIZE
FMOD.DSP_PITCHSHIFT_OVERLAP
FMOD.DSP_PITCHSHIFT_MAXCHANNELS
Values

**FMOD_DSP_PITCHSHIFT_PITCH**

(Type: float) - 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**

(Type: float) - 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**

(Type: float) - Removed. Do not use. FMOD now uses 4 overlaps and cannot be changed.

**FMOD_DSP_PITCHSHIFT_MAXCHANNELS**

(Type: float) - 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 smb@dspdimension.com.

'\texttt{maxchannels}' dictates the amount of memory allocated. By default, the \texttt{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 \texttt{ChannelGroup::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 \texttt{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.

**NOTE!** Not supported on PlayStation 3.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- ChannelGroup::addDSP
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_PROCESS_OPERATION

Operation type for `FMOD_DSP_PROCESS_CALLBACK`.

C/C++ Syntax

typedef enum {
    FMOD_DSP_PROCESS_PERFORM,
    FMOD_DSP_PROCESS_QUERY
} FMOD_DSP_PROCESS_OPERATION;

JavaScript Syntax

    FMOD.DSP_PROCESS_PERFORM
    FMOD.DSP_PROCESS_QUERY
Values

**FMOD_DSP_PROCESS_PERFORM**

Process the incoming audio in 'inbufferarray' and output to 'outbufferarray'.

**FMOD_DSP_PROCESS_QUERY**

The DSP is being queried for the expected output format and whether it needs to process audio or should be bypassed. The function should return **FMOD_OK**, or **FMOD_ERR_DSP_DONTPROCESS** or **FMOD_ERR_DSP_SILENCE** if audio can pass through unprocessed. See remarks for more. If audio is to be processed, 'outbufferarray' must be filled with the expected output format, channel count and mask.
Remarks

A process callback will be called twice per mix for a DSP unit. Once with the `FMOD_DSP_PROCESS_QUERY` command, then conditionally, `FMOD_DSP_PROCESS_PERFORM`. `FMOD_DSP_PROCESS_QUERY` is to be handled only by filling out the outputarray information, and returning a relevant return code. It should not really do any logic besides checking and returning one of the following codes:

- **FMOD_OK** - Meaning yes, it should execute the dsp process function with `FMOD_DSP_PROCESS_PERFORM`
- **FMOD_ERR_DSP_DONTPROCESS** - Meaning no, it should skip the process function and not call it with `FMOD_DSP_PROCESS_PERFORM`
- **FMOD_ERR_DSP_SILENCE** - Meaning no, it should skip the process function and not call it with `FMOD_DSP_PROCESS_PERFORM`, AND, tell the signal chain to follow that it is now idle, so that no more processing happens down the chain.

If audio is to be processed, 'outbufferarray' must be filled with the expected output format, channel count and mask. Mask can be 0.

`FMOD_DSP_PROCESS_PROCESS` is to be handled by reading the data from the input, processing it, and writing it to the output. Always write to the output buffer and fill it fully to avoid unpredictable audio output. Always return **FMOD_OK**, the return value is ignored from the process stage.
See Also

- FMOD_DSP_DESCRIPTION
Firelight Technologies FMOD Studio API
FMOD_DSP_RESAMPLER

List of interpolation types that the FMOD Studio software mixer supports.

C/C++ Syntax

typedef enum {
    FMOD_DSP_RESAMPLER_DEFAULT,
    FMOD_DSP_RESAMPLER_NOINTERP,
    FMOD_DSP_RESAMPLER_LINEAR,
    FMOD_DSP_RESAMPLER_CUBIC,
    FMOD_DSP_RESAMPLER_SPLINE,
    FMOD_DSP_RESAMPLER_MAX
} FMOD_DSP_RESAMPLER;

JavaScript Syntax

FMOD.DSP_RESAMPLER_DEFAULT
FMOD.DSP_RESAMPLER_NOINTERP
FMOD.DSP_RESAMPLER_LINEAR
FMOD.DSP_RESAMPLER_CUBIC
FMOD.DSP_RESAMPLER_SPLINE
FMOD.DSP_RESAMPLER_MAX
FMOD.DSP_RESAMPLER_FORCEINT
Values

*FMOD_DSP_RESAMPLER_DEFAULT*

Default interpolation method. Currently equal to *FMOD_DSP_RESAMPLER_LINEAR*.

*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::setAdvancedSettings` and the resamplerMethod member to tell FMOD the resampling quality you require for sample rate conversion during sound playback.
See Also

- System::setAdvancedSettings
- System::setAdvancedSettings
- FMOD_ADVANCEDSETTINGS
Firelight Technologies FMOD Studio API
FMOD_DSP_RETURN

Parameter types for the FMOD_DSP_TYPE_RETURN DSP.

C/C++ Syntax

typedef enum {
    FMOD_DSP_RETURN_ID,
    FMOD_DSP_RETURN_INPUT_SPEAKER_MODE
} FMOD_DSP_RETURN;

JavaScript Syntax

FMOD.DSP_RETURN_ID
FMOD.DSP_RETURN_INPUT_SPEAKER_MODE
Values

*FMOD_DSP_RETURN_ID*

(Type:int) - [r] ID of this Return DSP. Read-only. Default = -1.

*FMOD_DSP_RETURN_INPUT_SPEAKER_MODE*

(Type:int) - [r/w] Input speaker mode of this return. Default = *FMOD_SPEAKERMODE_DEFAULT*.
See Also

- DSP::setParameterInt
- DSP::getParameterInt
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_SEND

Parameter types for the FMOD_DSP_TYPE_SEND DSP.

C/C++ Syntax

typedef enum {
    FMOD_DSP_SEND_RETURNID,
    FMOD_DSP_SEND_LEVEL
} FMOD_DSP_SEND;

JavaScript Syntax

FMOD_DSP_SEND_RETURNID
FMOD_DSP_SEND_LEVEL
Values

\textit{FMOD\_DSP\_SEND\_RETURNID}

(Type:int) - ID of the Return DSP this send is connected to (integer values only). -1 indicates no connected Return DSP. Default = -1.

\textit{FMOD\_DSP\_SEND\_LEVEL}

(Type:float) - Send level. 0.0 to 1.0. Default = 1.0
See Also

- DSP::setParameterInt
- DSP::getParameterInt
- DSP::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_SFXREVERB

Parameter types for the FMOD_DSP_TYPE_SFXREVERB unit.

C/C++ Syntax

typedef enum {
    FMOD_DSP_SFXREVERB_DECAYTIME,
    FMOD_DSP_SFXREVERB_EARLYDELAY,
    FMOD_DSP_SFXREVERB_LATEDELAY,
    FMOD_DSP_SFXREVERB_HPREFERENCE,
    FMOD_DSP_SFXREVERB_HFDECAYRATIO,
    FMOD_DSP_SFXREVERB_DIFFUSION,
    FMOD_DSP_SFXREVERB_DENSITY,
    FMOD_DSP_SFXREVERB_LOWSHELFrequency,
    FMOD_DSP_SFXREVERB_LOWsHELF Ghana,
    FMOD_DSP_SFXREVERB_HIGHCUT,
    FMOD_DSP_SFXREVERB_EARLYLATEMIX,
    FMOD_DSP_SFXREVERB_WETLEVEL,
    FMOD_DSP_SFXREVERB_DRYLEVEL
} FMOD_DSP_SFXREVERB;

JavaScript Syntax

FMOD.DSP_SFXREVERB_DECAYTIME
FMOD.DSP_SFXREVERB_EARLYDELAY
FMOD.DSP_SFXREVERB_LATEDELAY
FMOD.DSP_SFXREVERB_HPREFERENCE
FMOD.DSP_SFXREVERB_HFDECAYRATIO
FMOD.DSP_SFXREVERB_DIFFUSION
FMOD.DSP_SFXREVERB_DENSITY
FMOD.DSP_SFXREVERB_HIGHCUT
FMOD.DSP_SFXREVERB_EARLYLATEMIX
FMOD.DSP_SFXREVERB_WETLEVEL
FMOD.DSP_SFXREVERB_DRYLEVEL
Values

FMOD_DSP_SFXREVERB_DECAYTIME

(Type:float) - Decay Time : Reverberation decay time at low-frequencies in milliseconds. Ranges from 100.0 to 20000.0. Default is 1500.

FMOD_DSP_SFXREVERB_EARLYDELAY

(Type:float) - Early Delay : Delay time of first reflection in milliseconds. Ranges from 0.0 to 300.0. Default is 20.

FMOD_DSP_SFXREVERB_LATEDELAY

(Type:float) - Reverb Delay : Late reverberation delay time relative to first reflection in milliseconds. Ranges from 0.0 to 100.0. Default is 40.

FMOD_DSP_SFXREVERB_HFREFERENCE

(Type:float) - HF Reference : Reference frequency for high-frequency decay in Hz. Ranges from 20.0 to 20000.0. Default is 5000.

FMOD_DSP_SFXREVERB_HFDECAYRATIO

(Type:float) - Decay HF Ratio : High-frequency decay time relative to decay time in percent. Ranges from 10.0 to 100.0. Default is 50.

FMOD_DSP_SFXREVERB_DIFFUSION

(Type:float) - Diffusion : Reverberation diffusion (echo density) in percent. Ranges from 0.0 to 100.0. Default is 100.

FMOD_DSP_SFXREVERB_DENSITY

(Type:float) - Density : Reverberation density (modal density) in percent. Ranges from 0.0 to 100.0. Default is 100.

FMOD_DSP_SFXREVERB_LOWSHELFFFREQUENCY
(Type:float) - Low Shelf Frequency : Transition frequency of low-shelf filter in Hz. Ranges from 20.0 to 1000.0. Default is 250.

**FMOD_DSP_SFXREVERB_LOWSHELFGAIN**

(Type:float) - Low Shelf Gain : Gain of low-shelf filter in dB. Ranges from -36.0 to 12.0. Default is 0.

**FMOD_DSP_SFXREVERB_HIGHCUT**

(Type:float) - High Cut : Cutoff frequency of low-pass filter in Hz. Ranges from 20.0 to 20000.0. Default is 20000.

**FMOD_DSP_SFXREVERB_EARLYLATEMIX**

(Type:float) - Early/Late Mix : Blend ratio of late reverb to early reflections in percent. Ranges from 0.0 to 100.0. Default is 50.

**FMOD_DSP_SFXREVERB_WETLEVEL**

(Type:float) - Wet Level : Reverb signal level in dB. Ranges from -80.0 to 20.0. Default is -6.

**FMOD_DSP_SFXREVERB_DRYLEVEL**

(Type:float) - Dry Level : Dry signal level in dB. Ranges from -80.0 to 20.0. Default is 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::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
- FMOD_REVERB_PRESETS
Firelight Technologies FMOD Studio API
FMOD_DSP_THREE_EQ

Parameter types for the FMOD_DSP_TYPE_THREE_EQ filter.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_THREE_EQ_LOWGAIN,
    FMOD_DSP_THREE_EQ_MIDGAIN,
    FMOD_DSP_THREE_EQ_HIGHGAIN,
    FMOD_DSP_THREE_EQ_LOWCROSSOVER,
    FMOD_DSP_THREE_EQ_HIGHCROSSOVER,
    FMOD_DSP_THREE_EQ_CROSSOVERSLOPE
} FMOD_DSP_THREE_EQ;
```

**JavaScript Syntax**

```javascript
FMOD.DSP.THREE_EQ_LOWGAIN
FMOD.DSP.THREE_EQ_MIDGAIN
FMOD.DSP.THREE_EQ_HIGHGAIN
FMOD.DSP.THREE_EQ_LOWCROSSOVER
FMOD.DSP.THREE_EQ_HIGHCROSSOVER
FMOD.DSP.THREE_EQ_CROSSOVERSLOPE
```
Values

FMOD_DSP_THREE_EQ_LOWGAIN
(Type:float) - Low frequency gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_THREE_EQ_MIDGAIN
(Type:float) - Mid frequency gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_THREE_EQ_HIGHGAIN
(Type:float) - High frequency gain in dB. -80.0 to 10.0. Default = 0.

FMOD_DSP_THREE_EQ_LOWCROSSOVER
(Type:float) - Low-to-mid crossover frequency in Hz. 10.0 to 22000.0. Default = 400.0.

FMOD_DSP_THREE_EQ_HIGHCROSSOVER
(Type:float) - Mid-to-high crossover frequency in Hz. 10.0 to 22000.0. Default = 4000.0.

FMOD_DSP_THREE_EQ_CROSSOVERSLOPE
(Type:int) - Crossover Slope. 0 = 12dB/Octave, 1 = 24dB/Octave, 2 = 48dB/Octave. Default = 1 (24dB/Octave).
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterInt
- DSP::getParameterInt
- FMOD_DSP_TYPE
- FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_TYPE
Firelight Technologies FMOD Studio API
Parameter values for the `FMOD_DSP_THREE_EQ_CROSSOVERSLOPE` parameter of the `FMOD_DSP_TYPE_THREE_EQ` DSP.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_12DB,
    FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_24DB,
    FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_48DB
} FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_TYPE;
```

**JavaScript Syntax**

```javascript
FMOD.DSP.THREE_EQ_CROSSOVERSLOPE_12DB
FMOD.DSP.THREE_EQ_CROSSOVERSLOPE_24DB
FMOD.DSP.THREE_EQ_CROSSOVERSLOPE_48DB
```
Values

`FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_12DB`

`FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_24DB`

`FMOD_DSP_THREE_EQ_CROSSOVERSLOPE_48DB`
See Also

- FMOD_DSP_THREE_EQ
Firelight Technologies FMOD Studio API
FMOD_DSP_TRANSCEIVER

Parameter types for the FMOD_DSP_TYPE_TRANSCEIVER filter.

C/C++ Syntax

typedef enum {
    FMOD_DSP_TRANSCEIVER_TRANSMIT,
    FMOD_DSP_TRANSCEIVER_GAIN,
    FMOD_DSP_TRANSCEIVER_CHANNEL,
    FMOD_DSP_TRANSCEIVER_TRANSMITSPEAKERMODE
} FMOD_DSP_TRANSCEIVER;

JavaScript Syntax

FMOD.DSP_TRANSCEIVER_TRANSMIT
FMOD.DSP_TRANSCEIVER_GAIN
FMOD.DSP_TRANSCEIVER_CHANNEL
FMOD.DSP_TRANSCEIVER_TRANSMITSPEAKERMODE
Values

**FMOD_DSP_TRANSCEIVER_TRANSMIT**

(Type:bool) - [r/w] - FALSE = Transceiver is a 'receiver' (like a return) and accepts data from a channel. TRUE = Transceiver is a 'transmitter' (like a send). Default = FALSE.

**FMOD_DSP_TRANSCEIVER_GAIN**

(Type:float) - [r/w] - Gain to receive or transmit at in dB. -80.0 to 10.0. Default = 0.

**FMOD_DSP_TRANSCEIVER_CHANNEL**

(Type:int) - [r/w] - Integer to select current global slot, shared by all Transceivers, that can be transmitted to or received from. 0 to 31. Default = 0.

**FMOD_DSP_TRANSCEIVER_TRANSMITSPEAKERMODE**

(Type:int) - [r/w] - Speaker mode (transmitter mode only). Specifies either 0 (Auto) Default = 0.
Remarks

The transceiver only transmits and receives to a global array of 32 channels. The transceiver can be set to receiver mode (like a return) and can receive the signal at a variable gain (\texttt{FMOD_DSP_TRANSCEIVER_GAIN}). The transceiver can also be set to transmit to a channel (like a send) and can transmit the signal with a variable gain (\texttt{FMOD_DSP_TRANSCEIVER_GAIN}).

The \texttt{FMOD_DSP_TRANSCEIVER_TRANSMITSPEAKERMODE} is only applicable to the transmission format, not the receive format. This means this parameter is ignored in 'receive mode'. This allows receivers to receive at the speaker mode of the user's choice. Receiving from a mono channel, is cheaper than receiving from a surround channel for example. The 3 speaker modes \texttt{FMOD_DSP_TRANSCEIVER_SPEAKERMODE_MONO}, \texttt{FMOD_DSP_TRANSCEIVER_SPEAKERMODE_STEREO}, \texttt{FMOD_DSP_TRANSCEIVER_SPEAKERMODE_SURROUND} are stored as separate buffers in memory for a transmitter channel. To save memory, use 1 common speaker mode for a transmitter.

The transceiver is double buffered to avoid desyncing of transmitters and receivers. This means there will be a 1 block delay on a receiver, compared to the data sent from a transmitter.

Multiple transmitters sending to the same channel will be mixed together.
See Also

- DSP::setParameterFloat
- DSP::getParameterFloat
- DSP::setParameterInt
- DSP::getParameterInt
- DSP::setParameterBool
- DSP::getParameterBool
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
Parameter types for the `FMOD_DSP_TRANSCEIVER_SPEAKERMODE` parameter for `FMOD_DSP_TYPE_TRANSCEIVER` effect.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_DSP_TRANSCEIVER_SPEAKERMODE_AUTO,
    FMOD_DSP_TRANSCEIVER_SPEAKERMODE_MONO,
    FMOD_DSP_TRANSCEIVER_SPEAKERMODE_STEREO,
    FMOD_DSP_TRANSCEIVER_SPEAKERMODE_SURROUND
} FMOD_DSP_TRANSCEIVER_SPEAKERMODE;
```

**JavaScript Syntax**

```javascript
FMOD.DSP_TRANSCEIVER_SPEAKERMODE_AUTO
FMOD.DSP_TRANSCEIVER_SPEAKERMODE_MONO
FMOD.DSP_TRANSCEIVER_SPEAKERMODE_STEREO
FMOD.DSP_TRANSCEIVER_SPEAKERMODE_SURROUND
```
Values

**FMOD_DSP_TRANSCEIVER_SPEAKERMODE_AUTO**

A transmitter will use whatever signal channel count coming in to the transmitter, to determine which speaker mode is allocated for the transceiver channel.

**FMOD_DSP_TRANSCEIVER_SPEAKERMODE_MONO**

A transmitter will always downmix to a mono channel buffer.

**FMOD_DSP_TRANSCEIVER_SPEAKERMODE_STEREO**

A transmitter will always upmix or downmix to a stereo channel buffer.

**FMOD_DSP_TRANSCEIVER_SPEAKERMODE_SURROUND**

A transmitter will always upmix or downmix to a surround channel buffer. Surround is the speaker mode of the system above stereo, so could be quad/surround/5.1/7.1.
**Remarks**

The speaker mode of a transceiver buffer (of which there are up to 32 of) is determined automatically depending on the signal flowing through the transceiver effect, or it can be forced. Use a smaller fixed speaker mode buffer to save memory.

Only relevant for transmitter dsps, as they control the format of the transceiver channel's buffer.

If multiple transceivers transmit to a single buffer in different speaker modes, it will allocate memory for each speaker mode. This uses more memory than a single speaker mode. If there are multiple receivers reading from a channel with multiple speaker modes, it will read them all and mix them together.

If the system's speaker mode is stereo or mono, it will not create a 3rd buffer, it will just use the mono/stereo speaker mode buffer.
See Also

- DSP::setParameterInt
- DSP::getParameterInt
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
**FMOD_DSP_TREMOLO**

Parameter types for the `FMOD_DSP_TYPE_TREMOLO` filter.

**C/C++ Syntax**

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;

**JavaScript Syntax**

`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`
Values

**FMOD_DSP_TREMOLO_FREQUENCY**
(Type: float) - LFO frequency in Hz. 0.1 to 20. Default = 5.

**FMOD_DSP_TREMOLO_DEPTH**
(Type: float) - Tremolo depth. 0 to 1. Default = 1.

**FMOD_DSP_TREMOLO_SHAPE**
(Type: float) - LFO shape morph between triangle and sine. 0 to 1. Default = 0.

**FMOD_DSP_TREMOLO_SKEW**
(Type: float) - Time-skewing of LFO cycle. -1 to 1. Default = 0.

**FMOD_DSP_TREMOLO_DUTY**
(Type: float) - LFO on-time. 0 to 1. Default = 0.5.

**FMOD_DSP_TREMOLO_SQUARE**
(Type: float) - Flatness of the LFO shape. 0 to 1. Default = 0.

**FMOD_DSP_TREMOLO_PHASE**
(Type: float) - Instantaneous LFO phase. 0 to 1. Default = 0.

**FMOD_DSP_TREMOLO_SPREAD**
(Type: float) - 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::setParameterFloat
- DSP::getParameterFloat
- FMOD_DSP_TYPE
Firelight Technologies FMOD Studio API
FMOD_DSP_TYPE

These definitions can be used for creating FMOD defined special effects or DSP units.

C/C++ Syntax

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_FADER,
    FMOD_DSP_TYPE_FLANGE,
    FMOD_DSP_TYPE_DISTORTION,
    FMOD_DSP_TYPE_NORMALIZE,
    FMOD_DSP_TYPE_LIMITER,
    FMOD_DSP_TYPE_PARAMEQ,
    FMOD_DSP_TYPE_PITCHSHIFT,
    FMOD_DSP_TYPE_CHORUS,
    FMOD_DSP_TYPE_VSTPLUGIN,
    FMOD_DSP_TYPE_WINAMPPPLUGIN,
    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_LADSPAPLUGIN,
    FMOD_DSP_TYPE_SEND,
    FMOD_DSP_TYPE_RETURN,
    FMOD_DSP_TYPE_HIGHPASS_SIMPLE,
    FMOD_DSP_TYPE_PAN,
    FMOD_DSP_TYPE_THREE_EQ,
    FMOD_DSP_TYPE_FFT,
    FMOD_DSP_TYPE_LOUDNESS_METER,
    FMOD_DSP_TYPE_ENVELOPEFOLLOWER,
    FMOD_DSP_TYPE_CONVOLUTIONREVERB,
    FMOD_DSP_TYPE_CHANNELMIX,
    FMOD_DSP_TYPE_TRANSCEIVER,
    FMOD_DSP_TYPE_OBJECTPAN,
} FMOD_DSP_TYPE;
```c
FMOD_DSP_TYPE_MULTIBAND_EQ,
FMOD_DSP_TYPE_MAX
} FMOD_DSP_TYPE;

JavaScript Syntax

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_FADER
FMOD.DSP_TYPE_FLANGE
FMOD.DSP_TYPE_DISTORTION
FMOD.DSP_TYPE_NORMALIZE
FMOD.DSP_TYPE_LIMITER
FMOD.DSP_TYPE_PARAMEQ
FMOD.DSP_TYPE_PITCHSHIFT
FMOD.DSP_TYPE_CHORUS
FMOD.DSP_TYPE_VSTPLUGIN
FMOD.DSP_TYPE_WINAMPP_PLUGIN
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_LADSPA_PLUGIN
FMOD.DSP_TYPE_SEND
FMOD.DSP_TYPE_RETURN
FMOD.DSP_TYPE_HIGHPASS_SIMPLE
FMOD.DSP_TYPE_PAN
FMOD.DSP_TYPE_THREE_EQ
FMOD.DSP_TYPE_FFT
FMOD.DSP_TYPE_LOUDNESS_METER
FMOD.DSP_TYPE_ENVELOPEFOLLOWER
FMOD.DSP_TYPE_CONVOLUTIONREVERB
FMOD.DSP_TYPE_CHANNELMIX
FMOD.DSP_TYPE_TRANSCEIVER
FMOD.DSP_TYPE_OBJECTPAN
FMOD.DSP_TYPE_MULTIBAND_EQ
FMOD.DSP_TYPE_MAX
FMOD.DSP_TYPE_FORCEINT
```
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. Deprecated and will be removed in a future release (see FMOD_DSP_LOWPASS remarks for alternatives).

*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. Deprecated and will be removed in a future release (see FMOD_DSP_HIGHPASS remarks for alternatives).

*FMOD_DSP_TYPE_ECHO*

This unit produces an echo on the sound and fades out at the desired rate.

*FMOD_DSP_TYPE_FADER*

This unit pans and scales the volume of a unit.
**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_LIMITER**
This unit limits the sound to a certain level.

**FMOD_DSP_TYPE_PARAMEQ**
This unit attenuates or amplifies a selected frequency range. Deprecated and will be removed in a future release (see **FMOD_DSP_PARAMEQ** remarks for alternatives).

**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_WINAMPPLUGIN**
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/unlinked 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. Deprecated and will be removed in a future release (see **FMOD_DSP_LOWPASS_SIMPLE** remarks for alternatives).

**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**

Unsupported / Deprecated.

**FMOD_DSP_TYPE_SEND**

This unit sends a copy of the signal to a return DSP anywhere in the DSP tree.

**FMOD_DSP_TYPE_RETURN**

This unit receives signals from a number of send DSPs.

**FMOD_DSP_TYPE_HIGHPASS_SIMPLE**

This unit filters sound using a simple highpass with no resonance, but has flexible cutoff and is fast. Deprecated and will be removed in a future release
(see `FMOD_DSP_HIGHPASS_SIMPLE` remarks for alternatives).

**FMOD_DSP_TYPE_PAN**

This unit pans the signal, possibly upmixing or downmixing as well.

**FMOD_DSP_TYPE_THREE_EQ**

This unit is a three-band equalizer.

**FMOD_DSP_TYPE_FFT**

This unit simply analyzes the signal and provides spectrum information back through getParameter.

**FMOD_DSP_TYPE_LOUDNESS_METER**

This unit analyzes the loudness and true peak of the signal.

**FMOD_DSP_TYPE_ENVELOPEFOLLOWER**

This unit tracks the envelope of the input/sidechain signal. Deprecated and will be removed in a future release.

**FMOD_DSP_TYPE_CONVOLUTIONREVERB**

This unit implements convolution reverb.

**FMOD_DSP_TYPE_CHANNELMIX**

This unit provides per signal channel gain, and output channel mapping to allow 1 multichannel signal made up of many groups of signals to map to a single output signal.

**FMOD_DSP_TYPE_TRANSCEIVER**

This unit 'sends' and 'receives' from a selection of up to 32 different slots. It is like a send/return but it uses global slots rather than returns as the destination. It also has other features. Multiple transceivers can receive from a single channel, or multiple transceivers can send to a single channel, or a combination of both.
**FMOD_DSP_TYPE_OBJECTPAN**

This unit sends the signal to a 3d object encoder like Dolby Atmos. Supports a subset of the **FMOD_DSP_TYPE_PAN** parameters.

**FMOD_DSP_TYPE_MULTIBAND_EQ**

This unit is a flexible five band parametric equalizer.

**FMOD_DSP_TYPE_MAX**

Maximum number of pre-defined DSP types.
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 Studio API
FMOD_ERRORCALLBACK_INSTANCETYPE

Used to distinguish the instance type passed into FMOD_ERROR_CALLBACK.

C/C++ Syntax

typedef enum {
    FMOD_ERRORCALLBACK_INSTANCETYPE_NONE,
    FMOD_ERRORCALLBACK_INSTANCETYPE_SYSTEM,
    FMOD_ERRORCALLBACK_INSTANCETYPE_CHANNEL,
    FMOD_ERRORCALLBACK_INSTANCETYPE_CHANNELGROUP,
    FMOD_ERRORCALLBACK_INSTANCETYPE_CHANNELCONTROL,
    FMOD_ERRORCALLBACK_INSTANCETYPE_SOUND,
    FMOD_ERRORCALLBACK_INSTANCETYPE_SOUNDGROUP,
    FMOD_ERRORCALLBACK_INSTANCETYPE_DSP,
    FMOD_ERRORCALLBACK_INSTANCETYPE_DSPCONNECTION,
    FMOD_ERRORCALLBACK_INSTANCETYPE_GEOMETRY,
    FMOD_ERRORCALLBACK_INSTANCETYPE_REVERB3D,
    FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_SYSTEM,
    FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_EVENTDESCRIPTION,
    FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_EVENTINSTANCE,
    FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_PARAMETERINSTANCE,
    FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_BUS,
    FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_VCA,
    FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_BANK,
    FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_COMMANDREPLAY
} FMOD_ERRORCALLBACK_INSTANCETYPE;

JavaScript Syntax

FMOD.ERRORCALLBACK_INSTANCETYPE_NONE
FMOD.ERRORCALLBACK_INSTANCETYPE_SYSTEM
FMOD.ERRORCALLBACK_INSTANCETYPE_CHANNEL
FMOD.ERRORCALLBACK_INSTANCETYPE_CHANNELGROUP
FMOD.ERRORCALLBACK_INSTANCETYPE_CHANNELCONTROL
FMOD.ERRORCALLBACK_INSTANCETYPE_SOUND
FMOD.ERRORCALLBACK_INSTANCETYPE_SOUNDGROUP
FMOD.ERRORCALLBACK_INSTANCETYPE_DSP
FMOD.ERRORCALLBACK_INSTANCETYPE_DSPCONNECTION
FMOD.ERRORCALLBACK_INSTANCETYPE_GEOMETRY
FMOD.ERRORCALLBACK_INSTANCETYPE_REVERB3D
FMOD.ERRORCALLBACK_INSTANCETYPE_STUDIO_SYSTEM
FMOD::ERRORCALLBACK_INSTANCETYPE_STUDIO_EVENTDESCRIPTION
FMOD::ERRORCALLBACK_INSTANCETYPE_STUDIO_EVENTINSTANCE
FMOD::ERRORCALLBACK_INSTANCETYPE_STUDIO_PARAMETERINSTANCE
FMOD::ERRORCALLBACK_INSTANCETYPE_STUDIO_BUS
FMOD::ERRORCALLBACK_INSTANCETYPE_STUDIO_VCA
FMOD::ERRORCALLBACK_INSTANCETYPE_STUDIO_BANK
FMOD::ERRORCALLBACK_INSTANCETYPE_STUDIO_COMMANDREPLAY
FMOD::ERRORCALLBACK_INSTANCETYPE_FORCEINT
Values

FMOD_ERRORCALLBACK_INSTANCETYPE_NONE
FMOD_ERRORCALLBACK_INSTANCETYPE_SYSTEM
FMOD_ERRORCALLBACK_INSTANCETYPE_CHANNEL
FMOD_ERRORCALLBACK_INSTANCETYPE_CHANNELGROUP
FMOD_ERRORCALLBACK_INSTANCETYPE_CHANNELCONTROL
FMOD_ERRORCALLBACK_INSTANCETYPE_SOUND
FMOD_ERRORCALLBACK_INSTANCETYPE_SOUNDGROUP
FMOD_ERRORCALLBACK_INSTANCETYPE_DSP
FMOD_ERRORCALLBACK_INSTANCETYPE_DSPCONNECTION
FMOD_ERRORCALLBACK_INSTANCETYPE_GEOMETRY
FMOD_ERRORCALLBACK_INSTANCETYPE_REVERB3D
FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_SYSTEM
FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_EVENTDESCRIPTION
FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_EVENTINSTANCE
FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_PARAMETERINSTANCE
FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_BUS
FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_VCA
FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_BANK
FMOD_ERRORCALLBACK_INSTANCETYPE_STUDIO_COMMANDREPLAY
Remarks

Cast the instance of FMOD_ERROR_CALLBACK to the appropriate class indicated by this enum.
Firelight Technologies FMOD Studio API
**FMOD_OPENSTATE**

These values describe what state a sound is in after `FMOD_NONBLOCKING` has been used to open it.

**C/C++ Syntax**

```c
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;
```

**JavaScript Syntax**

```javascript
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_FORCEINT
```
Values

FMOD_OPENSTATE_READY
Opened and ready to play.

FMOD_OPENSTATE_LOADING
Initial load in progress.

FMOD_OPENSTATE_ERROR
Failed to open - file not found, out of memory etc. See return value of Sound::getOpenState for what happened.

FMOD_OPENSTATE_CONNECTING
Connecting to remote host (internet sounds only).

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_OPENSTATESEEKING 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_OPENSTATESETPOSITION state and sound related commands will return FMOD_ERR_NOTREADY.
See Also

- `Sound::getOpenState`
- `FMOD_MODE`
Firelight Technologies FMOD Studio API
FMOD_OUTPUTTYPE

These output types are used with System::setOutput / System::getOutput, to choose which output method to use.

C/C++ Syntax

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_ASIO,
    FMOD_OUTPUTTYPE_PULSEAUDIO,
    FMOD_OUTPUTTYPE_ALSA,
    FMOD_OUTPUTTYPE_COREAUDIO,
    FMOD_OUTPUTTYPE_XAUDIO,
    FMOD_OUTPUTTYPE_PS3,
    FMOD_OUTPUTTYPE_AUDIOTRACK,
    FMOD_OUTPUTTYPE_OPENSLS,
    FMOD_OUTPUTTYPE_WIII,
    FMOD_OUTPUTTYPE_AUDIOOUT,
    FMOD_OUTPUTTYPE_AUDIO3D,
    FMOD_OUTPUTTYPE_ATMOS,
    FMOD_OUTPUTTYPE_WEBAUDIO,
    FMOD_OUTPUTTYPE_NNAUDIO,
    FMOD_OUTPUTTYPE_WINSONIC,
    FMOD_OUTPUTTYPE_MAX
} FMOD_OUTPUTTYPE;

JavaScript Syntax

FMOD.OUTPUTTYPE_AUTODETECT
FMOD.OUTPUTTYPE_UNKNOWN
FMOD.OUTPUTTYPE_NOSOUND
FMOD.OUTPUTTYPE_WAVWRITER
FMOD.OUTPUTTYPE_NOSOUND_NRT
FMOD.OUTPUTTYPE_WAVWRITER_NRT
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 - Perform all mixing but discard the final output.

**FMOD_OUTPUTTYPE_WAVWRITER**

All - Writes output to a .wav file.

**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**

Win - Direct Sound. (Default on Windows XP and below)

**FMOD_OUTPUTTYPE_WINMM**

Win - Windows Multimedia.

**FMOD_OUTPUTTYPE_WASAPI**

Win/WinStore/XboxOne - Windows Audio Session API. (Default on Windows
Vista and above, Xbox One and Windows Store Applications)

**FMOD_OUTPUTTYPE_ASIO**
Win - Low latency ASIO 2.0.

**FMOD_OUTPUTTYPE_PULSEAUDIO**
Linux - Pulse Audio. (Default on Linux if available)

**FMOD_OUTPUTTYPE_ALSA**
Linux - Advanced Linux Sound Architecture. (Default on Linux if PulseAudio isn't available)

**FMOD_OUTPUTTYPE_COREAUDIO**
Mac/iOS - Core Audio. (Default on Mac and iOS)

**FMOD_OUTPUTTYPE_XAUDIO**
Xbox 360 - XAudio. (Default on Xbox 360)

**FMOD_OUTPUTTYPE_PS3**
PS3 - Audio Out. (Default on PS3)

**FMOD_OUTPUTTYPE_AUDIOTRACK**
Android - Java Audio Track. (Default on Android 2.2 and below)

**FMOD_OUTPUTTYPE_OPENSL**
Android - OpenSL ES. (Default on Android 2.3 and above)

**FMOD_OUTPUTTYPE_WIIU**
Wii U - AX. (Default on Wii U)

**FMOD_OUTPUTTYPE_AUDIOOUT**
PS4/PSVita - Audio Out. (Default on PS4 and PS Vita)

`FMOD_OUTPUTTYPE_AUDIO3D`

PS4 - Audio3D.

`FMOD_OUTPUTTYPE_ATMOS`

Win - Dolby Atmos (WASAPI).

`FMOD_OUTPUTTYPE_WEB_AUDIO`

Web Browser - JavaScript webaudio output. (Default on JavaScript)

`FMOD_OUTPUTTYPE_NNAUDIO`

NX - NX nn::audio. (Default on NX)

`FMOD_OUTPUTTYPE_WINSONIC`

Win10 / XboxOne - Windows Sonic.

`FMOD_OUTPUTTYPE_MAX`

Maximum number of output types supported.
Remarks

To pass information to the driver when initializing fmod use the *extradrivernata* parameter in *System::init* for the following reasons.

- **FMOD_OUTPUTTYPE_WAVWRITER** - extradrivernata is a pointer to a char * file name that the wav writer will output to.
- **FMOD_OUTPUTTYPE_WAVWRITER_NRT** - extradrivernata is a pointer to a char * file name that the wav writer will output to.
- **FMOD_OUTPUTTYPE_DSOUND** - extradrivernata is cast to a HWND type, so that FMOD can set the focus on the audio for a particular window.
- **FMOD_OUTPUTTYPE_PS3** - extradrivernata is a pointer to a FMOD_PS3_EXTRADRIVERDATA struct. This can be found in fmodps3.h.
- **FMOD_OUTPUTTYPE_XAUDIO** - (Xbox360) extradrivernata is a pointer to a FMOD_360_EXTRADRIVERDATA struct. This can be found in fmodxbox360.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 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::init
- System::update
Firelight Technologies FMOD Studio API
**FMOD_PLUGINTYPE**

These are plugin types defined for use with the `System::getNumPlugins`, `System::getPluginInfo` and `System::unloadPlugin` functions.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_PLUGINTYPE_OUTPUT,
    FMOD_PLUGINTYPE_CODEC,
    FMOD_PLUGINTYPE_DSP,
    FMOD_PLUGINTYPE_MAX
} FMOD_PLUGINTYPE;
```

**JavaScript Syntax**

```javascript
FMOD.PLUGINTYPE_OUTPUT
FMOD.PLUGINTYPE_CODEC
FMOD.PLUGINTYPE_DSP
FMOD.PLUGINTYPE_MAX
FMOD.PLUGINTYPE_FORCEINT
```
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
Firelight Technologies FMOD Studio API
**FMOD_RESULT**

error codes. Returned from every function.

**C/C++ Syntax**

typedef enum {
    FMOD_OK,
    FMOD_ERR_BADCOMMAND,
    FMOD_ERR_CHANNEL_ALLOC,
    FMOD_ERR_CHANNEL_STOLEN,
    FMOD_ERR_DMA,
    FMOD_ERR_DSP_CONNECTION,
    FMOD_ERR_DSP_DONTPROCESS,
    FMOD_ERR_DSP_FORMAT,
    FMOD_ERR_DSP_INUSE,
    FMOD_ERR_DSP_NOTFOUND,
    FMOD_ERR_DSP_RESERVED,
    FMOD_ERR_DSP_SILENCE,
    FMOD_ERR_DSP_TYPE,
    FMOD_ERR_FILE_BAD,
    FMOD_ERR_FILE_COULDNOTSEEK,
    FMOD_ERR_FILE_DISKEJECTED,
    FMOD_ERR_FILE_EOF,
    FMOD_ERR_FILE_ENDOFDATA,
    FMOD_ERR_FILE_NOTFOUND,
    FMOD_ERR_FORMAT,
    FMOD_ERR_HEADER_MISMATCH,
    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_FLOAT,
    FMOD_ERR_INVALID_HANDLE,
    FMOD_ERR_INVALID_PARAM,
    FMOD_ERR_INVALID_POSITION,
    FMOD_ERR_INVALID_SPEAKER,
    FMOD_ERR_INVALID_SYNCPOINT,
    FMOD_ERR_INVALID_THREAD,
    FMOD_ERR_INVALID_VECTOR,
    FMOD_ERR_MAXAUDIBLE,
}
FMOD_OK
FMOD.ERR_BADCOMMAND
FMOD.ERR_CHANNEL_ALLOC
FMOD.ERR_CHANNEL_STOLEN
FMOD.ERR_DMA
FMOD.ERR_DSP_CONNECTION
FMOD.ERR_DSP_DONTPROCESS
FMOD.ERR_DSP_FORMAT
FMOD.ERR_DSP_INUSE
FMOD.ERR_DSP_NOTFOUND
FMOD.ERR_DSP_RESERVED
FMOD.ERR_DSP_SILENCE
FMOD.ERR_DSP_TYPE
FMOD.ERR_FILE_BAD
FMOD.ERR_FILE_COULDNOTSEEK
FMOD.ERR_FILE_DISKEJECTED
FMOD.ERR_FILE_EOF
FMOD.ERR_FILE_ENDOFDATA
FMOD.ERR_FILE_NOTFOUND
FMOD.ERR_FORMAT
FMOD.ERR_HEADER_MISMATCH
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_FLOAT
FMOD.ERR_INVALID_HANDLE
FMOD.ERR_INVALID_PARAM
FMOD.ERR_INVALID_POSITION
FMOD.ERR_INVALID_SPEAKER
FMOD.ERR_INVALID_SYNCPOINT
FMOD.ERR_INVALID_THREAD
FMOD.ERR_INVALID_VECTOR
FMOD.ERR_MAXAUDIBLE
FMOD.ERR_MEMORY
FMOD.ERR_MEMORY_CANTPOINT
FMOD.ERR_NEEDS3D
FMOD.ERR_NEEDSHARDWARE
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_FORMAT
FMOD.ERR_OUTPUT_INIT
FMOD.ERR_OUTPUT_NODRIVERS
FMOD.ERR_PLUGIN
FMOD.ERR_PLUGIN_MISSING
FMOD.ERR_PLUGIN_RESOURCE
FMOD.ERR_PLUGIN_VERSION
FMOD.ERR_RECORD
FMOD.ERR_REVERB_CHANNELGROUP
FMOD.ERR_REVERB_INSTANCE
FMOD.ERR_SUBSOUNDS
FMOD.ERR_SUBSOUND_ALLOCATED
FMOD.ERR_SUBSOUND_CANTMOVE
FMOD.ERR_TAGNOTFOUND
FMOD.ERR_TOO_MANY_CHANNELS
FMOD.ERR_TRUNCATED
FMOD.ERR_UNIMPLEMENTED
FMOD.ERR_UNINITIALIZED
FMOD.ERR_UNSUPPORTED
FMOD.ERR_VERSION
FMOD.ERR_EVENT_ALREADY_LOADED
FMOD.ERR_EVENT_LIVEUPDATE_BUSY
FMOD.ERR_EVENT_LIVEUPDATE_MISMATCH
FMOD.ERR_EVENT_LIVEUPDATE_TIMEOUT
FMOD.ERR_EVENT_NOTFOUND
FMOD.ERR_STUDIO_UNINITIALIZED
FMOD.ERR_STUDIO_NOT_LOADED
FMOD.ERR_INVALID_STRING
FMOD.ERR_ALREADY_LOCKED
FMOD.ERR_NOT_LOCKED
FMOD.ERR_RECORD_DISCONNECTED
FMOD.ERR_TOO_MANY_SAMPLES
FMOD.RESULT_FORCE_INT
Values

`FMOD_OK`

No errors.

`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_CHANNEL_ALLOC`

Error trying to allocate a channel.

`FMOD_ERR_CHANNEL_STOLEN`

The specified channel has been reused to play another sound.

`FMOD_ERR_DMA`

DMA Failure. See debug output for more information.

`FMOD_ERR_DSP_CONNECTION`

DSP connection error. Connection possibly caused a cyclic dependency or connected dsps with incompatible buffer counts.

`FMOD_ERR_DSP_DONTPROCESS`

DSP return code from a DSP process query callback. Tells mixer not to call the process callback and therefore not consume CPU. Use this to optimize the DSP graph.

`FMOD_ERR_DSP_FORMAT`

DSP Format error. A DSP unit may have attempted to connect to this network with the wrong format, or a matrix may have been set with the wrong size if the target unit has a specified channel map.
**FMOD_ERR_DSP_INUSE**

DSP is already in the mixer's DSP network. It must be removed before being reinserted or released.

**FMOD_ERR_DSP_NOTFOUND**

DSP connection error. Couldn't find the DSP unit specified.

**FMOD_ERR_DSP_RESERVED**

DSP operation error. Cannot perform operation on this DSP as it is reserved by the system.

**FMOD_ERR_DSP_SILENCE**

DSP return code from a DSP process query callback. Tells mixer silence would be produced from read, so go idle and not consume CPU. Use this to optimize the DSP graph.

**FMOD_ERR_DSP_TYPE**

DSP operation cannot be performed on a DSP of this type.

**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?).
**FMOD_ERR_FILE_ENDOFDATA**

End of current chunk reached while trying to read data.

**FMOD_ERR_FILE_NOTFOUND**

File not found.

**FMOD_ERR_FORMAT**

Unsupported file or audio format.

**FMOD_ERR_HEADER_MISMATCH**

There is a version mismatch between the FMOD header and either the FMOD Studio library or the FMOD Low Level library.

**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_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_THREAD**

Tried to call a function on a thread that is not supported.

**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_CREATECOMPRESSEDSAMPLE** was used.

**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.

**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_FORMAT**

Soundcard does not support the specified format.

**FMOD_ERR_OUTPUT_INIT**

Error initializing output device.

**FMOD_ERR_OUTPUT_NODRIVERS**

The output device has no drivers installed. If pre-init, FMOD_OUTPUT_NOSOUND is selected as the output mode. If post-init, the function just fails.

**FMOD_ERR_PLUGIN**

An unspecified error has been returned from a plugin.

**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_PLUGIN_VERSION**

A plugin was built with an unsupported SDK version.

**FMOD_ERR_RECORD**

An error occurred trying to initialize the recording device.

**FMOD_ERR_REVERB_CHANNELGROUP**

Reverb properties cannot be set on this channel because a parent channelgroup owns the reverb connection.

**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_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.

**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_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_TRUNCATED**

The retrieved string is too long to fit in the supplied buffer and has been truncated.

**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_VERSION**

The version number of this file format is not supported.

**FMOD_ERR_EVENT_ALREADY_LOADED**

The specified bank has already been loaded.

**FMOD_ERR_EVENT_LIVEUPDATE_BUSY**

The live update connection failed due to the game already being connected.

**FMOD_ERR_EVENT_LIVEUPDATE_MISMATCH**

The live update connection failed due to the game data being out of sync with the tool.
**FMOD_ERR_EVENT_LIVEUPDATE_TIMEOUT**
The live update connection timed out.

**FMOD_ERR_EVENT_NOTFOUND**
The requested event, bus or vca could not be found.

**FMOD_ERR_STUDIO_UNINITIALIZED**
The Studio::System object is not yet initialized.

**FMOD_ERR_STUDIO_NOT_LOADED**
The specified resource is not loaded, so it can't be unloaded.

**FMOD_ERR_INVALID_STRING**
An invalid string was passed to this function.

**FMOD_ERR_ALREADY_LOCKED**
The specified resource is already locked.

**FMOD_ERR_NOT_LOCKED**
The specified resource is not locked, so it can't be unlocked.

**FMOD_ERR_RECORD_DISCONNECTED**
The specified recording driver has been disconnected.

**FMOD_ERR_TOOMANYSAMPLES**
The length provided exceeds the allowable limit.
Firelight Technologies FMOD Studio API
FMOD_SOUNDGROUP_BEHAVIOR

These values are used with SoundGroup::setMaxAudibleBehavior to determine what happens when more sounds are played than are specified with SoundGroup::setMaxAudible.

C/C++ Syntax

typedef enum {
    FMOD_SOUNDGROUP_BEHAVIOR_FAIL,
    FMOD_SOUNDGROUP_BEHAVIOR_MUTE,
    FMOD_SOUNDGROUP_BEHAVIOR_STEALLOWEST,
    FMOD_SOUNDGROUP_BEHAVIOR_MAX
} FMOD_SOUNDGROUP_BEHAVIOR;

JavaScript Syntax

FMOD.SOUNDGROUP_BEHAVIOR_FAIL
FMOD.SOUNDGROUP_BEHAVIOR_MUTE
FMOD.SOUNDGROUP_BEHAVIOR_STEALLOWEST
FMOD.SOUNDGROUP_BEHAVIOR_MAX
FMOD.SOUNDGROUP_BEHAVIOR_FORCEINT
**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_STEALLELOWEST**

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 sound group behaviors.
Remarks

When using FMOD_SOUNDCOMPONENT_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`
Firelight Technologies FMOD Studio API
FMOD_SOUND_FORMAT

These definitions describe the native format of the hardware or software buffer that will be used.

C/C++ Syntax

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_BITSTREAM,
    FMOD_SOUND_FORMAT_MAX
} FMOD_SOUND_FORMAT;

JavaScript Syntax

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_BITSTREAM
FMOD.SOUND_FORMAT_MAX
FMOD.SOUND_FORMAT_FORCEINT
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_BITSTREAM**

Sound data is in its native compressed format.

**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 Studio API
**FMOD_SOUND_TYPE**

These definitions describe the type of song being played.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_SOUND_TYPE_UNKNOWN,
    FMOD_SOUND_TYPE_AIFF,
    FMOD_SOUND_TYPEASF,
    FMOD_SOUND_TYPE_DLS,
    FMOD_SOUND_TYPE_FLAC,
    FMOD_SOUND_TYPE_FSB,
    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_USER,
    FMOD_SOUND_TYPE_WAV,
    FMOD_SOUND_TYPE_XM,
    FMOD_SOUND_TYPE_XMA,
    FMOD_SOUND_TYPE_AUDIOQUEUE,
    FMOD_SOUND_TYPE_AT9,
    FMOD_SOUND_TYPE_VORBIS,
    FMOD_SOUND_TYPE_MEDIA_FOUNDATION,
    FMOD_SOUND_TYPE_MEDIACODEC,
    FMOD_SOUND_TYPE_FADPCM,
    FMOD_SOUND_TYPE_MAX
} FMOD_SOUND_TYPE;
```

**JavaScript Syntax**

```javascript
FMOD.SOUND_TYPE_UNKNOWN
FMOD.SOUND_TYPE_AIFF
FMOD.SOUND_TYPE_ASF
FMOD.SOUND_TYPE_DLS
FMOD.SOUND_TYPE_FLAC
FMOD.SOUND_TYPE_FSB
FMOD.SOUND_TYPE_IT
FMOD.SOUND_TYPE_MIDI
```
Values

*FMOD_SOUND_TYPE_UNKNOWN*
3rd party / unknown plugin format.

*FMOD_SOUND_TYPE_AIFF*
AIFF.

*FMOD_SOUND_TYPEASF*
Microsoft Advanced Systems Format (ie WMA/ASF/WMV).

*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_IT*
Impulse Tracker.

*FMOD_SOUND_TYPE_MIDI*
MIDI.

*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_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_AUDIOQUEUE*

iPhone hardware decoder, supports AAC, ALAC and MP3.

*FMOD_SOUND_TYPE_AT9*

PS4 / PSVita ATRAC 9 format
**FMOD_SOUND_TYPE_VORBIS**

Vorbis

**FMOD_SOUND_TYPE_MEDIA_FOUNDATION**

Windows Store Application built in system codecs

**FMOD_SOUND_TYPE_MEDIACODEC**

Android MediaCodec

**FMOD_SOUND_TYPE_FADPCM**

FMOD Adaptive Differential Pulse Code Modulation

**FMOD_SOUND_TYPE_MAX**

Maximum number of sound types supported.
See Also

- Sound::getFormat
Firelight Technologies FMOD Studio API
**FMOD_SPEAKER**

Assigns an enumeration for a speaker index.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_SPEAKER_FRONT_LEFT,
    FMOD_SPEAKER_FRONT_RIGHT,
    FMOD_SPEAKER_FRONT_CENTER,
    FMOD_SPEAKER_LOW_FREQUENCY,
    FMOD_SPEAKER_SURROUND_LEFT,
    FMOD_SPEAKER_SURROUND_RIGHT,
    FMOD_SPEAKER_BACK_LEFT,
    FMOD_SPEAKER_BACK_RIGHT,
    FMOD_SPEAKER_TOP_FRONT_LEFT,
    FMOD_SPEAKER_TOP_FRONT_RIGHT,
    FMOD_SPEAKER_TOP_BACK_LEFT,
    FMOD_SPEAKER_TOP_BACK_RIGHT,
    FMOD_SPEAKER_MAX
} FMOD_SPEAKER;
```

**JavaScript Syntax**

```javascript
FMOD.SPEAKER_FRONT_LEFT
FMOD.SPEAKER_FRONT_RIGHT
FMOD.SPEAKER_FRONT_CENTER
FMOD.SPEAKER_LOW_FREQUENCY
FMOD.SPEAKER_SURROUND_LEFT
FMOD.SPEAKER_SURROUND_RIGHT
FMOD.SPEAKER_BACK_LEFT
FMOD.SPEAKER_BACK_RIGHT
FMOD.SPEAKER_MAX
FMOD.SPEAKER_FORCEINT
```
Values

**FMOD_SPEAKER_FRONT_LEFT**
The front left speaker

**FMOD_SPEAKER_FRONT_RIGHT**
The front right speaker

**FMOD_SPEAKER_FRONT_CENTER**
The front center speaker

**FMOD_SPEAKER_LOW_FREQUENCY**
The LFE or 'subwoofer' speaker

**FMOD_SPEAKER_SURROUND_LEFT**
The surround left (usually to the side) speaker

**FMOD_SPEAKER_SURROUND_RIGHT**
The surround right (usually to the side) speaker

**FMOD_SPEAKER_BACK_LEFT**
The back left speaker

**FMOD_SPEAKER_BACK_RIGHT**
The back right speaker

**FMOD_SPEAKER_TOP_FRONT_LEFT**
The top front left speaker

**FMOD_SPEAKER_TOP_FRONT_RIGHT**
The top front right speaker

*FMOD_SPEAKER_TOP_BACK_LEFT*

The top back left speaker

*FMOD_SPEAKER_TOP_BACK_RIGHT*

The top back right speaker

*FMOD_SPEAKER_MAX*

Maximum number of speaker types supported.
See Also

- System::setSpeakerPosition
- System::getSpeakerPosition
Firelight Technologies FMOD Studio API
FMOD_SPEAKERMODE

These are speaker types defined for use with the System::setSoftwareFormat command.

C/C++ Syntax

typedef enum {
    FMOD_SPEAKERMODE_DEFAULT,
    FMOD_SPEAKERMODE_RAW,
    FMOD_SPEAKERMODE_MONO,
    FMOD_SPEAKERMODE_STEREO,
    FMOD_SPEAKERMODE_QUAD,
    FMOD_SPEAKERMODE_SURROUND,
    FMOD_SPEAKERMODE_5POINT1,
    FMOD_SPEAKERMODE_7POINT1,
    FMOD_SPEAKERMODE_7POINT1POINT4,
    FMOD_SPEAKERMODE_MAX
} FMOD_SPEAKERMODE;

JavaScript Syntax

FMOD.SPEAKERMODE_DEFAULT
FMOD.SPEAKERMODE_RAW
FMOD.SPEAKERMODE_MONO
FMOD.SPEAKERMODE_STEREO
FMOD.SPEAKERMODE_QUAD
FMOD.SPEAKERMODE_SURROUND
FMOD.SPEAKERMODE_5POINT1
FMOD.SPEAKERMODE_7POINT1
FMOD.SPEAKERMODE_MAX
FMOD.SPEAKERMODE_FORCEINT
Values

**FMOD_SPEAKERMODE_DEFAULT**

Default speaker mode for the chosen output mode which will resolve after `System::init`.

**FMOD_SPEAKERMODE_RAW**

Assume there is no special mapping from a given channel to a speaker, channels map 1:1 in order. Use `System::setSoftwareFormat` to specify the speaker count.

**FMOD_SPEAKERMODE_MONO**

1 speaker setup (monaural).

**FMOD_SPEAKERMODE_STEREO**

2 speaker setup (stereo) front left, front right.

**FMOD_SPEAKERMODE_QUAD**

4 speaker setup (4.0) front left, front right, surround left, surround right.

**FMOD_SPEAKERMODE_SURROUND**

5 speaker setup (5.0) front left, front right, center, surround left, surround right.

**FMOD_SPEAKERMODE_5POINT1**

6 speaker setup (5.1) front left, front right, center, low frequency, surround left, surround right.

**FMOD_SPEAKERMODE_7POINT1**

8 speaker setup (7.1) front left, front right, center, low frequency, surround left, surround right, back left, back right.

**FMOD_SPEAKERMODE_7POINT1POINT4**
12 speaker setup (7.1.4) front left, front right, center, low frequency, surround left, surround right, back left, back right, top front left, top front right, top back left, top back right.

**FMOD_SPEAKERMODE_MAX**

Maximum number of speaker modes supported.
Remarks

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::setPanLevels do not work.
Speaker levels must be manually set with Channel::setPanMatrix.

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::setPanMatrix. Channel::setPanLevels 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::setPanMatrix`.
`Channel::setPanLevels` 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, surround left and a surround 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::setPanMatrix`.
• `Channel::setPanLevels` works but rear left, rear right, center and lfe are ignored.

**FMOD_SPEAKERMODE_SURROUND**
------------------------
This mode is for 5 speaker arrangements that have a left/right/center/surround left/surround 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::setPanMatrix`.
• `Channel::setPanLevels` works but rear left / rear right are ignored.

**FMOD_SPEAKERMODE_5POINT1**
----------------------------------------
This mode is for 5.1 speaker arrangements that have a left/right/center/surround left/surround 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::setPanMatrix`.
- `Channel::setPanLevels` works but rear left / rear right are ignored.

**FMOD_SPEAKERMODE_7POINT1**

This mode is for 7.1 speaker arrangements that have a left/right/center/surround left/surround right/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::setPanMatrix`.
- `Channel::setPanLevels` works and every parameter is used to set the balance of a sound in any speaker.
See Also

- `System::setSoftwareFormat`
- `System::getSoftwareFormat`
- `DSP::setChannelFormat`
Firelight Technologies FMOD Studio API
**FMOD_TAGDATATYPE**

List of data types that can be returned by `Sound::getTag`

### C/C++ Syntax

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;

### JavaScript Syntax

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_FORCEINT
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_CDOCTYPE`

`FMOD_TAGDATATYPE_MAX`

Maximum number of tag datatypes supported.
See Also

- `Sound::getTag`
Firelight Technologies FMOD Studio API
**FMOD_TAGTYPE**

List of tag types that could be stored within a sound. These include id3 tags, metadata from netstreams and vorbis/asf data.

**C/C++ Syntax**

```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;
```

**JavaScript Syntax**

```javascript
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_FORCEINT
```
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 Studio API
Studio API

Classes Functions
Structures
Defines
Enumerations
Firelight Technologies FMOD Studio API
Classes

System  EventDescription
EventInstance
CueInstance
ParameterInstance
Bus
VCA
Bank
CommandReplay
Firelight Technologies FMOD Studio API
Studio::System

The main system object for FMOD Studio.
Functions

Studio::System::create Studio::System::flushCommands
Studio::System::flushSampleLoading
Studio::System::getAdvancedSettings
Studio::System::getBank
Studio::System::getBankByID
Studio::System::getBankCount
Studio::System::getBankList
Studio::System::getBufferUsage
Studio::System::getBus
Studio::System::getBusByID
Studio::System::getCPUUsage
Studio::System::getEvent
Studio::System::getEventByID
Studio::System::getListenerAttributes
Studio::System::getListenerWeight
Studio::System::getLowLevelSystem
Studio::System::getNumListeners
Studio::System::getSoundInfo
Studio::System::getUserData
Studio::System::getVCA
Studio::System::getVCAByID
Studio::System::initialize
Studio::System::loadBankCustom
Studio::System::loadBankFile
Studio::System::loadBankMemory
Studio::System::loadCommandReplay
Studio::System::lookupID
Studio::System::lookupPath
Studio::System::registerPlugin
Studio::System::release
Studio::System::resetBufferUsage
Studio::System::setAdvancedSettings
Studio::System::setCallback
Studio::System::setListenerAttributes
Studio::System::setListenerWeight
Studio::System::setNumListeners
Studio::System::setUserData
Studio::System::startCommandCapture
Studio::System::stopCommandCapture
Studio::System::unloadAll
Studio::System::unregisterPlugin
Studio::System::update
Remarks

Initializing the FMOD Studio System object will also initialise the low level System object.
See Also

- Studio::System::create
- Studio::System::initialize
Firelight Technologies FMOD Studio API
Studio::System::create

Creates a Studio System object. This must be called before you do anything else.

C++ Syntax

```cpp
static FMOD_RESULT Studio::System::create(
    Studio::System **system,
    unsigned int headerVersion
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_Create(
    FMOD_STUDIO_SYSTEM **system,
    unsigned int headerVersion
);
```

C# Syntax

```csharp
static RESULT Studio.System.create(
    out System studiostystem
);
```

JavaScript Syntax

```javascript
static System.create(
    system, // writes value to system.val
    headerversion
);
```
Parameters

`system`

Address of a variable to receive the new Studio System object.

`headerVersion`

The expected FMOD Studio API version, to ensure the library matches the headers. For the C API, pass in FMOD_VERSION. For the C++ API, it defaults to FMOD_VERSION, so you don't need to pass it in explicitly.
Return Values

If the function succeeds then the return value 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 also creates the internal Low Level System object. Call
Studio::System::release to free the Studio System.

System::setSoftwareFormat must be called on the Low Level System object with
speakermode corresponding to the project's output format if there is a possibility
of the output audio device not matching the project's format. Any differences
between the project format and the Low Level System's speakermode will cause
the mix to sound wrong.

NOTE: Calls to Studio::System::create and Studio::System::release are not
thread-safe. Do not call these functions simultaneously from multiple threads at
once.
See Also

- Studio::System::initialize
- Studio::System::release
- Studio::System::getLowLevelSystem
- System::setSoftwareFormat
Firelight Technologies FMOD Studio API
Studio::System::flushCommands

Waits until all pending commands have been executed.

C++ Syntax

FMOD_RESULT Studio::System::flushCommands();

C Syntax

FMOD_RESULT FMOD_Studio_System_FlushCommands(FMOD_STUDIO_SYSTEM *sys

C# Syntax

RESULT Studio.System.flushCommands();

JavaScript Syntax

System.flushCommands();
Return Values

If the function succeeds then the return value 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 Studio has been initialized in asynchronous mode, this function will block the calling thread until the pending command buffer is empty and until all non-blocking bank loads have completed.

This is equivalent to calling `Studio::System::update` and then sleeping until the asynchronous thread has finished executing all pending commands.

You can use this function to ensure that Studio has finished executing all pending commands, for example, when measuring memory usage after calling bank unload.

This function does not wait for sample data loading to complete.
See Also

- Studio::System::initialize
- Studio::System::update
- Studio::System::flushSampleLoading
Firelight Technologies FMOD Studio API
Studio::System::flushSampleLoading

Waits until all sample loading and unloading has completed.

C++ Syntax

FMOD_RESULT Studio::System::flushSampleLoading();

C Syntax

FMOD_RESULT FMOD_Studio_System_FlushSampleLoading(FMOD_STUDIO_SYSTEM

C# Syntax

RESULT Studio.System.flushSampleLoading();

JavaScript Syntax

System.flushSampleLoading();
Return Values

If the function succeeds then the return value 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 block the calling thread until all pending sample loading and unloading has completed.

Sample loading can be caused by `Studio::Bank::loadSampleData`, `Studio::EventDescription::loadSampleData`, and `Studio::EventDescription::createInstance`.

You can use this function to ensure that Studio has finished executing sample loading, for example measuring memory or disk usage.

**NOTE!** This function may stall for a long time if other threads are continuing to issue calls to load and unload sample data, such continually creating new event instances.
See Also

- Studio::System::flushCommands
- Studio::Bank::loadSampleData
- Studio::EventDescription::loadSampleData
Firelight Technologies FMOD Studio API
Studio::System::getAdvancedSettings

Retrieves the advanced settings assigned to the studio system object.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::getAdvancedSettings(
    FMOD_STUDIO_ADVANCEDSETTINGS *settings
);```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_GetAdvancedSettings(
    FMOD_STUDIO_SYSTEM *system,
    FMOD_STUDIO_ADVANCEDSETTINGS *settings
);```

C# Syntax

```csharp
RESULT Studio.System.getAdvancedSettings(
    out ADVANCEDSETTINGS settings
);```

JavaScript Syntax

```javascript
System.getAdvancedSettings(
    settings // writes value to settings.val
);```
Parameters

settings
Address of a variable to receive the contents of the
FMOD_STUDIO_ADVANCEDSETTINGS structure specified by the user.
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

The cbSize field must be set to sizeof(FMOD_STUDIO_ADVANCEDSETTINGS) before calling this function.
See Also

- FMOD_STUDIO_ADVANCEDSETTINGS
- Studio::System::setAdvancedSettings
Firelight Technologies FMOD Studio API
**Studio::System::getBank**

Retrieves an already loaded Bank object by path, filename or ID string.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getBank(
    const char *path,
    Studio::Bank **bank
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetBank(
    FMOD_STUDIO_SYSTEM *system,
    const char *path,
    FMOD_STUDIO_BANK **bank
);
```

**C# Syntax**

```cs
RESULT Studio.System.getBank(
    string path,
    out Bank bank
);
```

**JavaScript Syntax**

```javascript
System.getBank(
    path,
    bank
) // writes value to bank.val
```
Parameters

path

The bank path, filename, or the ID string that identifies the bank.

bank

Address of a variable to receive the Bank 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

If a bank path or ID is passed in, this is equivalent to calling Studio::System::lookupID or Studio::parseID and then calling Studio::System::GetBankByID. If a bank filename is passed in, it is first converted to a bank path.

The path field can be the bank path such as 'bank:/Weapons', the bank filename such as '../Desktop/Weapons.bank', or the ID such as '{793cddb6-7fa1-4e06-b805-4c74c0fd625b}'. 
See Also

- Studio::parseID
- Studio::System::lookupID
- Studio::System::getBankByID
Firelight Technologies FMOD Studio API
**Studio::System::getBankByID**

Retrieves an already loaded Bank object by ID.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getBankByID(
    const FMOD_GUID *id,
    Studio::Bank **bank
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetBankByID(
    FMOD_STUDIO_SYSTEM *system,
    const FMOD_GUID *id,
    FMOD_STUDIO_BANK **bank
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getBankByID(
    Guid guid,
    out Bank bank
);
```

**JavaScript Syntax**

```javascript
System.getBankByID(
    id,
    bank // writes value to bank.val
);
```
Parameters

id
   The 128-bit GUID which identifies the bank.

bank
   Address of a variable to receive the Bank 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 bank ID can be looked up using \texttt{Studio::System::lookupID} or parsed from a string using \texttt{Studio::parseID}.
See Also

- Studio::parseID
- Studio::System::lookupID
- Studio::System::getBank
Firelight Technologies FMOD Studio API
Studio::System::getBankCount

Retrieves the number of loaded banks.

C++ Syntax

FMOD_RESULT Studio::System::getBankCount(
    int *count
);

C Syntax

FMOD_RESULT FMOD_Studio_System_GetBankCount(
    FMOD_STUDIO_SYSTEM *system,
    int *count
);

C# Syntax

RESULT Studio.System.getBankCount(
    out int count
);

JavaScript Syntax

System.getBankCount(
    count
    // writes value to count.val
);
Parameters

count
  Address of a variable to receive the number of loaded Banks.
Return Values

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

Used in conjunction with Studio::System::getBankList to enumerate the banks.
See Also

- Studio::System::getBankList
Firelight Technologies FMOD Studio API
Studio::System::getBankList

Retrieves the loaded Banks.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getBankList(
    Studio::Bank **array,
    int capacity,
    int *count
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetBankList(
    FMOD_STUDIO_SYSTEM *system,
    FMOD_STUDIO_BANK **array,
    int capacity,
    int *count
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getBankList(
    out Bank[] array
);
```

**JavaScript Syntax**

```javascript
System.getBankList(
    array,       // writes value to array.val
    capacity,    // writes value to count.val
    count
);
```
Parameters

array
   An array of memory allocated by the user.
capacity
   The capacity of the array passed in as the first parameter
count
   Address of a variable to receive the number of Banks written to the array
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

Used in conjunction with Studio::System::getBankCount to enumerate the banks.
See Also

- Studio::System::getBankCount
Firelight Technologies FMOD Studio API
Studio::System::getBufferUsage

Retrieves information about various memory buffers used by FMOD Studio.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::getBufferUsage(
    FMOD_STUDIO_BUFFER_USAGE *usage
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_GetBufferUsage(
    FMOD_STUDIO_SYSTEM *system,
    FMOD_STUDIO_BUFFER_USAGE *usage
);
```

C# Syntax

```csharp
RESULT Studio.System.getBufferUsage(
    out BUFFER_USAGE usage
);
```

JavaScript Syntax

```javascript
System.getBufferUsage(
    usage // writes value to usage.val
);
```
Parameters

usage
   Address of a variable to receive the performance information.
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

Stall count and time values are cumulative. They can be reset by calling `Studio::System::resetBufferUsage`.

Stalls due to the studio command queue can be avoided by calling `Studio::System::setAdvancedSettings` to set a larger queue size.
See Also

- FMOD_STUDIO_BUFFER_USAGE
- Studio::System::resetBufferUsage
Firelight Technologies FMOD Studio API
Studio::System::getBus

Retrieves a bus by path or ID string.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getBus(
    const char *path,
    Studio::Bus **bus
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetBus(  
    FMOD_STUDIO_SYSTEM *system,
    const char *path,
    FMOD_STUDIO_BUS **bus
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getBus(
    string path,
    out Bus bus
);
```

**JavaScript Syntax**

```javascript
System.getBus(
    path,
    bus // writes value to bus.val
);
```
Parameters

path
   The bus path or the ID string that identifies the bus.

bus
   Address of a variable to receive the bus 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 allows you to retrieve a handle for any bus in the global mixer.

The path field can be the full bus path, such as 'bus:/SFX/Ambience', or the ID, such as '{d9982c58-a056-4e6c-b8e3-883854b4bff}'.

Note that path lookups will only succeed if the strings bank has been loaded.
See Also

- Studio::parseID
- Studio::System::lookupID
- Studio::System::getBusByID
Firelight Technologies FMOD Studio API
Studio::System::getBusBylD

Retrieves a bus by ID.

**C++ Syntax**

FMOD_RESULT Studio::System::getBusBylD(
    const FMOD_GUID *id,
    Studio::Bus **bus
);

**C Syntax**

FMOD_RESULT FMOD_Studio_System_GetBusBylD(
    FMOD_STUDIO_SYSTEM *system,
    const FMOD_GUID *id,
    FMOD_STUDIO_BUS **bus
);

**C# Syntax**

RESULT Studio.System.getBusBylD(
    Guid guid,
    out Bus bus
);

**JavaScript Syntax**

System.getBusBylD(
    id,
    bus // writes value to bus.val
);
Parameters

id
   The 128-bit GUID which identifies the bus.

bus
   Address of a variable to receive the bus 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

This function allows you to retrieve a handle for any bus in the global mixer. The ID can be looked up using Studio::System::lookupID or parsed from a string using Studio::parseID.
See Also

- Studio::parseID
- Studio::System::lookupID
- Studio::System::getBus
Firelight Technologies FMOD Studio API
Studio::System::getCPUUsage

Retrieves performance information for FMOD Studio and low level systems.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getCPUUsage(
    FMOD_STUDIO_CPU_USAGE *usage
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetCPUUsage(
    FMOD_STUDIO_SYSTEM *system,
    FMOD_STUDIO_CPU_USAGE *usage
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getCPUUsage(
    out CPU_USAGE usage
);
```

**JavaScript Syntax**

```javascript
System.getCPUUsage(
    usage // writes value to usage.val
);
```
Parameters

usage
   Address of a variable to receive the performance 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_STUDIO_CPU_USAGE
Firelight Technologies FMOD Studio API
**Studio::System::getEvent**

Retrieves an EventDescription by path or ID string.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getEvent(
    const char *path,
    Studio::EventDescription **event
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetEvent(
    FMOD_STUDIO_SYSTEM *system,
    const char *path,
    FMOD_STUDIO_EVENTDESCRIPTION **event
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getEvent(
    string path,
    out EventDescription _event
);
```

**JavaScript Syntax**

```javascript
System.getEvent(
    path,
    event // writes value to event.val
);
```
Parameters

path
   The path or the ID string that identifies the event or snapshot.

event
   Address of a variable to receive the EventDescription 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 allows you to retrieve a handle to any loaded event description.

The path field can be the full event path, such as 'event:/UI/Cancel' or 'snapshot:/IngamePause', or the ID in the form '{2a3e48e6-94fc-4363-9468-33d2dd4d7b00}'.

Note that path lookups will only succeed if the strings bank has been loaded.
See Also

- `Studio::parseID`
- `Studio::System::lookupID`
- `Studio::System::getEventByID`
- `Studio::EventDescription::isSnapshot`
- `Studio::EventDescription::createInstance`
Firelight Technologies FMOD Studio API
**Studio::System::getEventByID**

Retrieves an EventDescription by ID.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getEventByID(
    const FMOD_GUID *id,
    Studio::EventDescription **event
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetEventByID(
    FMOD_STUDIO_SYSTEM *system,
    const FMOD_GUID *id,
    FMOD_STUDIO_EVENTDESCRIPTION **event
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getEventByID(
    Guid guid,
    out EventDescription _event
);
```

**JavaScript Syntax**

```javascript
System.getEventByID(
    id,
    event // writes value to event.val
);
```
Parameters

id
The 128-bit GUID which identifies the event or snapshot.

event
Address of a variable to receive the EventDescription 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 ID can be looked up using `Studio::System::lookupID` or parsed from a string using `Studio::parseID`.
See Also

- Studio::parseID
- Studio::System::lookupID
- Studio::System::getEvent
- Studio::EventDescription::isSnapshot
- Studio::EventDescription::createInstance
Firelight Technologies FMOD Studio API
Studio::System::getListenerAttributes

Retrieves the position, velocity and orientation of the 3D sound listener.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::getListenerAttributes(
    int listener,
    FMOD_3D_ATTRIBUTES *attributes
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_GetListenerAttributes(
    FMOD_STUDIO_SYSTEM *system,
    int listener,
    FMOD_3D_ATTRIBUTES *attributes
);
```

C# Syntax

```csharp
RESULT Studio.System.getListenerAttributes(
    int listener,
    out _3D_ATTRIBUTES attributes
);
```

JavaScript Syntax

```javascript
System.getListenerAttributes(
    listener,
    attributes // writes value to attributes.val
);
```
Parameters

listener
   Listener index. Specify 0 if there is only 1 listener.

attributes
   Address of a variable to receive the 3D attributes for the listener. See FMOD_3D_ATTRIBUTES.
Return Values

If the function succeeds then the return 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

- Studio::EventInstance::get3DAttributes
- Studio::System::getNumListeners
- Studio::System::setListenerAttributes
Firelight Technologies FMOD Studio API
Studio::System::getListenerWeight

Gets the listener weighting, which allows listeners to fade in and out.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::getListenerWeight(
    int listener,
    float *weight
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_GetListenerWeight(
    FMOD_STUDIO_SYSTEM *system,
    int listener,
    float *weight
);
```

C# Syntax

```csharp
RESULT Studio.System.getListenerWeight(
    int listener,
    out float weight
);
```

JavaScript Syntax

```javascript
System.addListenerWeight(
    listener,
    weight // writes value to weight.val
);
```
Parameters

listener
    Listener index.

weight
    Address of a variable to receive the weighting 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

- Studio::System::setListenerWeight
- Studio::System::setNumListeners
- Studio::System::getListenerAttributes
Firelight Technologies FMOD Studio API
**Studio::System::getLowLevelSystem**

Retrieves the Studio System's internal Low Level System object for access to the Low Level API.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getLowLevelSystem(
    FMOD::System **lowLevelSystem
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetLowLevelSystem(
    FMOD_STUDIO_SYSTEM *system,
    FMOD_SYSTEM **lowLevelSystem
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getLowLevelSystem(
    out FMOD.System system
);
```

**JavaScript Syntax**

```javascript
System.getLowLevelSystem(
    system // writes value to system.val
);
```
Parameters

lowLevelSystem
Return Values

If the function succeeds then the return value 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 call Low Level configuration functions before initialization.
See Also

- System::setFileSystem
- System::setDSPBufferSize
- System::setSoftwareChannels
- System::setSoftwareFormat
- System::setAdvancedSettings
- System::setCallback
Firelight Technologies FMOD Studio API
Studio::System::getNumListeners

Gets the number of listeners that have been set into in the 3D sound scene.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getNumListeners(
    int *numlisteners
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetNumListeners(
    FMOD_STUDIO_SYSTEM *system,
    int *numlisteners
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getNumListeners(
    out int numlisteners
);
```

**JavaScript Syntax**

```javascript
System.getNumListeners(  
    numlisteners // writes value to numlisteners.v
);
```
**Parameters**

`numlisteners`

Number of listeners that have been 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

- Studio::System::setNumListeners
Firelight Technologies FMOD Studio API
Studio::System::getSoundInfo

Retrieves information for loading a sound from an audio table.

C++ Syntax

FMOD_RESULT Studio::System::getSoundInfo(
  const char *key,
  FMOD_STUDIO_SOUND_INFO *info
);

C Syntax

FMOD_RESULT FMOD_Studio_System_GetSoundInfo(
  FMOD_STUDIO_SYSTEM *system,
  const char *key,
  FMOD_STUDIO_SOUND_INFO *info
);

C# Syntax

RESULT Studio.System.getSoundInfo(
  string key,
  out SOUND_INFO info
);

JavaScript Syntax

System.getSoundInfo(
  key,
  info // writes value to info.val
);
Parameters

key
  The key that identifies the sound.
info
  Address of a variable to receive the sound loading 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.
Remarks

The **FMOD_STUDIO_SOUND_INFO** structure contains information to be passed to `System::createSound` (which will create a parent sound), along with a subsound index to be passed to `Sound::getSubSound` once the parent sound is loaded.

The user is expected to call `System::createSound` with the given information. It is up to the user to OR in any desired loading flags, such as `FMOD_CREATESTREAM`, `FMOD_CREATECOMPRESSESAMPLE` or `FMOD_NONBLOCKING`. The default loading is the equivalent of `FMOD_CREATESAMPLE`, which will load the sample data and decompress into memory.

When the banks have been loaded via `Studio::System::LoadBankMemory`, the mode will be returned as `FMOD_OPENMEMORY_POINT`. This won't work with the default `FMOD_CREATESAMPLE` mode. For memory banks, you should add in the `FMOD_CREATECOMPRESSESAMPLE` or `FMOD_CREATESTREAM` flag, or alternatively remove `FMOD_OPENMEMORY_POINT` and change it to `FMOD_OPENMEMORY` if you really want to decompress the sample out of the memory bank into a new allocation.

Example code:

```c
const char* audio_table_name = "some name";
FMOD::Sound* sound = NULL;
FMOD_STUDIO_SOUND_INFO info = {};

// Look up the entry
ERRCHECK(studioSystem->getSoundInfo(audio_table_name, &info));

// Load compressed sample data
info.mode |= FMOD_CREATECOMPRESSESAMPLE;

// Create the sound
ERRCHECK(lowLevelSystem->createSound(info.name_or_data, info.mode, &

// We end up with a parent sound, and info.subsoundIndex. The subsound can be given to a programmer
See Also

- FMOD_STUDIO_SOUND_INFO
- FMOD_STUDIO_EVENT_CALLBACK
- FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES
- System::createSound
Firelight Technologies FMOD Studio API
**Studio::System::getUserData**

Retrieves the user data that is set on the system.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::getUserData(
    void **userData
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_GetUserData(
    FMOD_STUDIO_SYSTEM *system,
    void **userData
);
```

**C# Syntax**

```csharp
RESULT Studio.System.getUserData(
    out IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
System.getUserData( 
    userdata
    // writes value to userdata.val
);
Parameters

userData
    Address of a variable to receive the 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.
See Also

- Studio::System::setUserData
Firelight Technologies FMOD Studio API
Studio::System::getVCA

Retrieves a VCA by path or ID string.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::getVCA(
    const char *path,
    Studio::VCA **vca
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_GetVCA(
    FMOD_STUDIO_SYSTEM *system,
    const char *path,
    FMOD_STUDIO_VCA **vca
);
```

C# Syntax

```csharp
RESULT Studio.System.getVCA(
    string path,
    out VCA vca
);
```

JavaScript Syntax

```javascript
System.getVCA(
    path,
    vca // writes value to vca.val
);
```
**Parameters**

**path**
The VCA path or the ID string that identifies the VCA.

**vca**
Address of a variable to receive the VCA 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 allows you to retrieve a handle for any VCA in the global mixer.

The path field can be the full VCA path, such as 'vca:/MyVCA', or the ID, such as '{d9982c58-a056-4e6c-b8e3-883854b4bffb}'.

Note that path lookups will only succeed if the strings bank has been loaded.
See Also

- Studio::parseID
- Studio::System::lookupID
- Studio::System::getVCAByID
Firelight Technologies FMOD Studio API
Studio::System::getVCAByID

Retrieves a VCA by ID.

C++ Syntax

FMOD_RESULT Studio::System::getVCAByID(
    const FMOD_GUID *id,
    Studio::VCA **vca
);

C Syntax

FMOD_RESULT FMOD_Studio_System_GetVCAByID(
    FMOD_STUDIO_SYSTEM *system,
    const FMOD_GUID *id,
    FMOD_STUDIO_VCA **vca
);

C# Syntax

RESULT Studio.System.getVCAByID(
    Guid guid,
    out VCA vca
);

JavaScript Syntax

System.getVCAByID(
    id,
    vca // writes value to vca.val
);
**Parameters**

**id**

The 128-bit GUID which identifies the VCA.

**vca**

Address of a variable to receive the VCA 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 allows you to retrieve a handle for any VCA in the global mixer. The ID can be looked up using Studio::System::lookupID or parsed from a string using Studio::parseID.
See Also

- Studio::parseID
- Studio::System::lookupID
- Studio::System::getVCA
Firelight Technologies FMOD Studio API
Studio::System::initialize

Initializes the Studio System, the Low Level System, and the sound device.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::initialize(
    int maxchannels,
    FMOD_STUDIO_INITFLAGS studioflags,
    FMOD_INITFLAGS flags,
    void *extradriverdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_Initialize(
    FMOD_STUDIO_SYSTEM *system,
    int maxchannels,
    FMOD_STUDIO_INITFLAGS studioflags,
    FMOD_INITFLAGS flags,
    void *extradriverdata
);
```

**C# Syntax**

```csharp
RESULT Studio.System.initialize(
    int maxchannels,
    INITFLAGS studioFlags,
    FMOD.INITFLAGS flags,
    IntPtr extradriverdata
);
```

**JavaScript Syntax**

```javascript
System.initialize(
    maxchannels,
    studioflags,
    flags,
    extradriverdata
);
```
Parameters

maxchannels
The maximum number of channels to be used in FMOD. These 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 real voices.

studioflags
See FMOD_STUDIO_INITFLAGS. This can be a selection of flags bitwise OR'ed together to change the behaviour of the Studio System.

flags
See FMOD_INITFLAGS. This can be a selection of flags bitwise OR'ed together to change the behaviour of the Low Level System.

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 \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 function should be called at startup time, after creating a Studio System with `Studio::System::create`.

By default Studio is initialized in asynchronous mode where all API calls are automatically deferred for execution on a separate Studio thread. You can use `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE` to disable this behaviour and have all commands executed inside `Studio::System::update`. 
See Also

- `Studio::System::create`
Firelight Technologies FMOD Studio API
Studio::System::loadBankCustom

Loads a Studio event bank using custom read callbacks.

**C++ Syntax**

```c++
FMOD_RESULT Studio::System::loadBankCustom(
    const FMOD_STUDIO_BANK_INFO *info,
    FMOD_STUDIO_LOAD_BANK_FLAGS flags,
    Studio::Bank **bank
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_LoadBankCustom(
    FMOD_STUDIO_SYSTEM *system,
    const FMOD_STUDIO_BANK_INFO *info,
    FMOD_STUDIO_LOAD_BANK_FLAGS flags,
    FMOD_STUDIO_BANK **bank
);
```

**C# Syntax**

```csharp
RESULT Studio.System.loadBankCustom(
    BANK_INFO info,
    LOAD_BANK_FLAGS flags,
    out Bank bank
);
```

**JavaScript Syntax**

```javascript
System.loadBankCustom(
    info,
    flags,
    bank // writes value to bank.val
);
```
Parameters

info
  Information for loading the bank.

flags
  Flags to control bank loading.

bank
  Address of a variable to receive the Bank 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

By default this function will block until the file load finishes, and then return the `FMOD_RESULT` indicating the result. If there was a file error for blocking bank loads, then the bank variable will be returned as NULL.

Using the `FMOD_STUDIO_LOAD_BANK_NONBLOCKING` flag will cause the bank to be loaded asynchronously. In that case it will always return `FMOD_OK` and a valid bank handle. File errors for asynchronous banks can be detected by calling `Studio::Bank::getLoadingState`. Failed asynchronous banks should be released by calling `Studio::Bank::unload`. 
See Also

- FMOD_STUDIO_BANK_INFO
- FMOD_STUDIO_LOAD_BANK_FLAGS
- Studio::System::loadBankFile
- Studio::System::loadBankMemory
- Studio::Bank::getLoadingState
- Studio::Bank::unload
Firelight Technologies FMOD Studio API
Studio::System::loadBankFile

Loads a Studio event bank from a file.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::loadBankFile(
    const char *filename,
    FMOD_STUDIO_LOAD_BANK_FLAGS flags,
    Studio::Bank **bank
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_LoadBankFile(
    FMOD_STUDIO_SYSTEM *system,
    const char *filename,
    FMOD_STUDIO_LOAD_BANK_FLAGS flags,
    FMOD_STUDIO_BANK **bank
);
```

**C# Syntax**

```csharp
RESULT Studio.System.loadBankFile(
    string name,
    LOAD_BANK_FLAGS flags,
    out Bank bank
);
```

**JavaScript Syntax**

```javascript
System.loadBankFile(
    filename,
    flags,
    bank);              // writes value to bank.val
```
Parameters

filename
   Name of the file on disk.
flags
   Flags to control bank loading.
bank
   Address of a variable to receive the Bank 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

By default this function will block until the file load finishes, and then return the `FMOD_RESULT` indicating the result. If there was a file error for blocking bank loads, then the bank variable will be returned as NULL.

Using the `FMOD_STUDIO_LOAD_BANK_NONBLOCKING` flag will cause the bank to be loaded asynchronously. In that case it will always return `FMOD_OK` and a valid bank handle. File errors for asynchronous banks can be detected by calling `Studio::Bank::getLoadingState`. Failed asynchronous banks should be released by calling `Studio::Bank::unload`. 
See Also

- FMOD_STUDIO_LOAD_BANK_FLAGS
- Studio::System::loadBankMemory
- Studio::System::loadBankCustom
- Studio::Bank::getLoadingState
- Studio::Bank::unload
Firelight Technologies FMOD Studio API
Studio::System::loadBankMemory

Loads a Studio event bank from memory.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::loadBankMemory(
    const char *buffer,
    int length,
    FMOD_STUDIO_LOAD_MEMORY_MODE mode,
    FMOD_STUDIO_LOAD_BANK_FLAGS flags,
    Studio::Bank **bank
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_LoadBankMemory(
    FMOD_STUDIO_SYSTEM *system,
    const char *buffer,
    int length,
    FMOD_STUDIO_LOAD_MEMORY_MODE mode,
    FMOD_STUDIO_LOAD_BANK_FLAGS flags,
    FMOD_STUDIO_BANK **bank
);
```

C# Syntax

```csharp
RESULT Studio.System.loadBankMemory(
    byte[] buffer,
    LOAD_BANK_FLAGS flags,
    out Bank bank
);
```

JavaScript Syntax

```javascript
System.loadBankMemory(
    buffer,
    length,
    mode,
    flags,
    bank // writes value to bank.val
);
```
**Parameters**

**buffer**
Memory buffer to load from. This should be 32-byte aligned if mode is `FMOD_STUDIO_LOAD_MEMORY_POINT`.

**length**
Length of the memory buffer.

**mode**
Loading mode to use.

**flags**
Flags to control bank loading.

**bank**
Address of a variable to receive the Bank 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

Specifying mode `FMOD_STUDIO_LOAD_MEMORY_POINT` will POINT to your memory rather than FMOD allocating its own buffers and duplicating it internally. When using this mode, the buffer must be aligned to `FMOD_STUDIO_LOAD_MEMORY_ALIGNMENT` and the lifetime of the memory must persist until the bank has been queued for unload and then the unload has been performed. You can ensure the memory is not being freed prematurely by only freeing it after receiving the `FMOD_STUDIO_SYSTEM_CALLBACK_BANK_UNLOAD` callback.

By default this function will block until the file load finishes, and then return the `FMOD_RESULT` indicating the result. If there was a file error for blocking bank loads, then the bank variable will be returned as NULL.

Using the `FMOD_STUDIO_LOAD_BANK_NONBLOCKING` flag will cause the bank to be loaded asynchronously. In that case it will always return `FMOD_OK` and a valid bank handle. File errors for asynchronous banks can be detected by calling `Studio::Bank::getLoadingState`. Failed asynchronous banks should be released by calling `Studio::Bank::unload`. 
See Also

- `FMOD_STUDIO_LOAD_MEMORY_MODE`
- `FMOD_STUDIO_LOAD_BANK_FLAGS`
- Studio::System::loadBankFile
- Studio::System::loadBankCustom
- Studio::System::setCallback
- Studio::Bank::getLoadingState
- Studio::Bank::unload
Firelight Technologies FMOD Studio API
Studio::System::loadCommandReplay

Playback Studio commands that have previously been recorded to file.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::loadCommandReplay(
    const char *filename,
    FMOD_STUDIO_COMMANDREPLAY_FLAGS flags
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_LoadCommandReplay(
    FMOD_STUDIO_SYSTEM *system,
    const char *filename,
    FMOD_STUDIO_COMMANDREPLAY_FLAGS flags
);
```

C# Syntax

```csharp
RESULT Studio.System.loadCommandReplay(
    string path,
    COMMANDREPLAY_FLAGS flags,
    out CommandReplay replay
);
```

JavaScript Syntax

```javascript
System.loadCommandReplay(
    filename,
    flags,
    playback // writes value to playback.val
);
```
Parameters

filename
The filename to load the command replay file from.

flags
Flags that control the command replay.
Return Values

If the function succeeds then the return 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_STUDIO_COMMANDREPLAY_FLAGS`
- `Studio::System::startCommandCapture`
- `Studio::System::stopCommandCapture`
- `Studio::CommandReplay::start`
- `Studio::CommandReplay::stop`
- `Studio::CommandReplay::release`
Firelight Technologies FMOD Studio API
Studio::System::lookupID

Retrieves the ID for a bank, event, snapshot, bus or VCA.

**C++ Syntax**

```c++
FMOD_RESULT Studio::System::lookupID(
    const char *path,
    FMOD_GUID *id
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_LookupID(
    FMOD_STUDIO_SYSTEM *system,
    const char *path,
    FMOD_GUID *id
);
```

**C# Syntax**

```csharp
RESULT Studio.System.lookupID(
    string path,
    out Guid guid
);
```

**JavaScript Syntax**

```javascript
System.lookupID(
    path,
    id // writes value to id.val
);
```
**Parameters**

*path*

The path to the object as shown in FMOD Studio.

*id*

Address of a variable to receive the 128-bit GUID.
Return Values

If the function succeeds then the return value 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 return \texttt{FMOD\_ERR\_EVENT\_NOTFOUND} unless string data for the requested object is loaded (by loading the "Master Bank.strings.bank" file).

The path can be retrieved from FMOD Studio using the "Copy Path" context menu command. Paths must be in the following format:

- "event:/UI/Cancel"
- "snapshot:/IngamePause"
- "bus:/SFX/Ambience"
- "vca:/Mega Strip"
- "bank:/Vehicles"
See Also

- Studio::System::getEvent
- Studio::System::getBus
- Studio::System::getVCA
- Studio::System::getBank
- Studio::System::lookupPath
Firelight Technologies FMOD Studio API
Studio::System::lookupPath

Retrieves the path for a bank, event, snapshot, bus or VCA.

C++ Syntax

FMOD_RESULT Studio::System::lookupPath(
    const FMOD_GUID *id,
    char *path,
    int size,
    int *retrieved
);

C Syntax

FMOD_RESULT FMOD_Studio_System_LookupPath(
    FMOD_STUDIO_SYSTEM *system,
    const FMOD_GUID *id,
    char *path,
    int size,
    int *retrieved
);

C# Syntax

RESULT Studio.System.lookupPath(
    Guid guid,
    out string path
);

JavaScript Syntax

System.lookupPath(
    id,
    path,                   // writes value to path.val
    size,
    retrieved              // writes value to retrieved.val
);
**Parameters**

**id**
The 128-bit GUID which identifies the bank, event, snapshot, bus or VCA.

**path**
Address of a buffer to receive the path. Specify 0 or NULL to ignore.

**size**
Size of the path buffer in bytes. Required if *path* parameter is not NULL.

**retrieved**
Address of a variable to receive the size of the retrieved path in bytes, including the terminating null character. 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 will return `FMOD_ERR_EVENT_NOTFOUND` unless string data for the requested object is loaded (by loading the "Master Bank.strings.bank" file).

If the retrieved path is too long to fit in the supplied buffer, it will be truncated and this function will return `FMOD_ERR_TRUNCATED`. Use the `retrieved` parameter to obtain the minimum buffer size required to hold the full path.

**JavaScript only:**

Note: For the "path" parameter, the maximum string length is 512.
See Also

- Studio::System::lookupID
Firelight Technologies FMOD Studio API
Studio::System::registerPlugin

 Registers a third party plugin DSP for use by events loaded by the Studio API.

 **C++ Syntax**

```c++
FMOD_RESULT Studio::System::registerPlugin(
    const FMOD_DSP_DESCRIPTION *description
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_RegisterPlugin(
    FMOD_STUDIO_SYSTEM *system,
    const FMOD_DSP_DESCRIPTION *description
);
```

**C# Syntax**

```csharp
RESULT Studio.System.registerPlugin(
);
```

**JavaScript Syntax**

```javascript
System.registerPlugin(
    description
);
```
**Parameters**

**description**
The description of the DSP. See [FMOD_DSP_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

- Studio::System::unregisterPlugin
Firelight Technologies FMOD Studio API
Studio::System::release

Shuts down and frees the Studio System.

C++ Syntax

FMOD_RESULT Studio::System::release();

C Syntax

FMOD_RESULT FMOD_Studio_System_Release(FMOD_STUDIO_SYSTEM *system);

C# Syntax

RESULT Studio.System.release();

JavaScript Syntax

System.release();
Return Values

If the function succeeds then the return value 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 Studio System object and everything created under it.

NOTE: Calls to Studio::System::create and Studio::System::release are not thread-safe. Do not call these functions simultaneously from multiple threads at once.
See Also

- Studio::System::create
Firelight Technologies FMOD Studio API
Studio::System::resetBufferUsage

Resets information about memory buffers used by FMOD Studio.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::resetBufferUsage();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_ResetBufferUsage(FMOD_STUDIO_SYSTEM *
```

**C# Syntax**

```csharp
RESULT Studio.System.resetBufferUsage();
```

**JavaScript Syntax**

```javascript
System.resetBufferUsage();
```
Return Values

If the function succeeds then the return value 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 resets the stall count, stall time, and peak usage fields of the `FMOD_STUDIO_BUFFER_USAGE` struct.
See Also

- `FMOD_STUDIO_BUFFER_USAGE`
- `Studio::System::getBufferUsage`
Firelight Technologies FMOD Studio API
Studio::System::setAdvancedSettings

Sets advanced features like configuring memory and cpu usage.

C++ Syntax

FMOD_RESULT Studio::System::setAdvancedSettings(
    FMOD_STUDIO_ADVANCEDSETTINGS *settings
);

C Syntax

FMOD_RESULT FMOD_Studio_System_SetAdvancedSettings(
    FMOD_STUDIO_SYSTEM *system,
    FMOD_STUDIO_ADVANCEDSETTINGS *settings
);

C# Syntax

RESULT Studio.System.setAdvancedSettings(
    ADVANCEDSETTINGS settings
);

JavaScript Syntax

System.setAdvancedSettings(
    settings
);
Parameters

settings
  Pointer to `FMOD_STUDIO_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.
Remarks

This function should be called before Studio initialization. The cbSize field must be set to sizeof(FMOD_STUDIO_ADVANCEDSETTINGS) before calling this function.
See Also

- FMOD_STUDIO_ADVANCEDSETTINGS
- Studio::System::getAdvancedSettings
- Studio::System::initialize
Firelight Technologies FMOD Studio API
Studio::System::setCallback

Sets a callback to hook into various informational events.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::setCallback(FMOD_STUDIO_SYSTEM_CALLBACK callback,
                                        FMOD_STUDIO_SYSTEM_CALLBACK_TYPE callbackmask);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_SetCallback(FMOD_STUDIO_SYSTEM *system,
                                          FMOD_STUDIO_SYSTEM_CALLBACK callback,
                                          FMOD_STUDIO_SYSTEM_CALLBACK_TYPE callbackmask);
```

C# Syntax

```csharp
RESULT Studio.System.setCallback(
                                 SYSTEM_CALLBACK callback,
                                 SYSTEM_CALLBACK_TYPE callbackmask = SYSTEM_CALLBACK_TYPE.ALL);
```

JavaScript Syntax

```javascript
System.setCallback(
                   callback,
                   callbackmask);
```
Parameters

callback
    Pointer to a callback to receive the callback when it happens.
callbackmask
    A bitfield specifying which callback types are required. Masking out some
callback types can help avoid a flood of irrelevant callbacks being
triggered. Defaults to `FMOD_STUDIO_SYSTEM_CALLBACK_ALL`. 
Return Values

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

Some callbacks can occur asynchronously depending on Studio initialisation flags.

Example:

```cpp
FMOD_RESULT F_CALLBACK MySystemCallback(FMOD_STUDIO_SYSTEM *system,
FMOD_RESULT SetupEventCallback(Studio::System *system)
{
    return system->setCallback(MySystemCallback, FMOD_STUDIO_SYSTEM
}
See Also

- FMOD_STUDIO_SYSTEM_CALLBACK
- FMOD_STUDIO_SYSTEM_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
**Studio::System::setListenerAttributes**

Sets the position, velocity and orientation of the 3D sound listener.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::setListenerAttributes(
    int listener,
    FMOD_3D_ATTRIBUTES *attributes
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_SetListenerAttributes(
    FMOD_STUDIO_SYSTEM *system,
    int listener,
    FMOD_3D_ATTRIBUTES *attributes
);
```

**C# Syntax**

```csharp
RESULT Studio.System.setListenerAttributes(
    int listener,
    _3D_ATTRIBUTES attributes
);
```

**JavaScript Syntax**

```javascript
System.setListenerAttributes(
    listener,
    attributes
);
```
Parameters

**listener**
Listener index. Specify 0 if there is only 1 listener.

**attributes**
The 3D attributes for the listener. See [FMOD_3D_ATTRIBUTES](#).
Return Values

If the function succeeds then the return value 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 should be called every game tick, or at least whenever the listener moves.
See Also

- Studio::EventInstance::set3DAttributes
- Studio::System::setNumListeners
- Studio::System::setListenerWeight
- Studio::System::getListenerAttributes
Firelight Technologies FMOD Studio API
Studio::System::setListenerWeight

Sets the listener weighting, allowing listeners to fade in and out.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::setListenerWeight(
    int listener,
    float weight
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_SetListenerWeight(
    FMOD_STUDIO_SYSTEM *system,
    int listener,
    float weight
);
```

**C# Syntax**

```csharp
RESULT Studio.System.setListenerWeight(
    int listener,
    float weight
);
```

**JavaScript Syntax**

```javascript
System.setListenerWeight(
    listener,
    weight
);
```
**Parameters**

**listener**
- Listener index.

**weight**
- The weighting value from 0 to 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

Listener weighting is a factor that determines how much the listener contributes to the mix. It is taken into account for 3D panning, doppler, and the automatic distance event parameter. A listener with a weight of 0 has no effect on the mix.

Listener weighting can be used to fade in and out multiple listeners. For example to do a crossfade, an additional listener can be created with a weighting of 0 that ramps up to 1 while the old listener weight is ramped down to 0. After the crossfade is finished the number of listeners can be reduced to 1 again.

The sum of all the listener weights should add up to at least 1. It is a user error to set all listener weights to 0.
See Also

- Studio::System::getListenerWeight
- Studio::EventInstance::set3DAttributes
- Studio::System::setNumListeners
- Studio::System::getListenerAttributes
Firelight Technologies FMOD Studio API
**Studio::System::setNumListeners**

Sets the number of listeners in the 3D sound scene. This function is useful mainly for split-screen game purposes.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::setNumListeners(
    int numlisteners
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_SetNumListeners(
    FMOD_STUDIO_SYSTEM *system,
    int numlisteners
);
```

**C# Syntax**

```csharp
RESULT Studio.System.setNumListeners(
    int numlisteners
);
```

**JavaScript Syntax**

```javascript
System.setNumListeners(
    numlisteners
);
```
Parameters

numlisteners
   Number of listeners in the scene. Valid values are from 1 to \texttt{FMOD_MAX_LISTENERS} 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 FMOD uses a 'closest sound to the listener' method to determine what should be heard.
See Also

- Studio::System::getNumListeners
- Studio::System::setListenerAttributes
- Studio::System::setListenerWeight
- FMOD_MAX_LISTENERS
Firelight Technologies FMOD Studio API
**Studio::System::setUserData**

Sets arbitrary user data on the system.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::setUserData(
    void *userData
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_SetUserData(
    FMOD_STUDIO_SYSTEM *system,
    void *userData
);
```

**C# Syntax**

```csharp
RESULT Studio.System.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
System.setUserData(
    userdata
);
```
Parameters

userData
    Address of user data to store within the system.
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 function is primarily used in case the user wishes to 'attach' data to an FMOD object.
See Also

- [Studio::System::getUserData](#)
Firelight Technologies FMOD Studio API
Studio::System::startCommandCapture

Start recording all Studio commands to a file with the given path.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::startCommandCapture(
    const char *filename,
    FMOD_STUDIO_COMMANDCAPTURE_FLAGS flags
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_StartCommandCapture(
    FMOD_STUDIO_SYSTEM *system,
    const char *filename,
    FMOD_STUDIO_COMMANDCAPTURE_FLAGS flags
);
```

C# Syntax

```csharp
RESULT Studio.System.startCommandCapture(
    string path,
    COMMANDCAPTURE_FLAGS flags
);
```

JavaScript Syntax

```javascript
System.startCommandCapture(
    filename,
    flags
);
```
Parameters

filename
   The filename where the write the command replay file.

flags
   Flags that control command capturing.
Return Values

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

Commands can be captured and then replayed for debug purposes.

Unless the `FMOD_STUDIO_COMMANDCAPTURE_SKIP_INITIAL_STATE` flag is specified, the command capture will first record the set of all banks and event instances that currently exist.
See Also

- `FMOD_STUDIO_COMMANDCAPTURE_FLAGS`
- `Studio::System::stopCommandCapture`
- `Studio::System::loadCommandReplay`
Firelight Technologies FMOD Studio API
Studio::System::stopCommandCapture

Stop recording Studio commands.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::stopCommandCapture();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_StopCommandCapture(FMOD_STUDIO_SYSTEM
```

**C# Syntax**

```csharp
RESULT Studio.System.stopCommandCapture();
```

**JavaScript Syntax**

```javascript
System.stopCommandCapture();
```
Return Values

If the function succeeds then the return 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

- Studio::System::startCommandCapture
- Studio::System::loadCommandReplay
Firelight Technologies FMOD Studio API
Studio::System::unloadAll

Unloads all currently loaded banks.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::System::unloadAll();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_System_UnloadAll(FMOD_STUDIO_SYSTEM *system);
```

**C# Syntax**

```csharp
RESULT Studio.System.unloadAll();
```

**JavaScript Syntax**

```javascript
System.unloadAll();
```
Return Values

If the function succeeds then the return 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 Studio API
Studio::System::unregisterPlugin

Unregisters a previously registered third party plugin DSP.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::unregisterPlugin(
    const char *name
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_UnregisterPlugin(
    FMOD_STUDIO_SYSTEM *system,
    const char *name
);
```

C# Syntax

```csharp
RESULT Studio.System.unregisterPlugin(
);
```

JavaScript Syntax

```javascript
System.unregisterPlugin(
    name
);
```
Parameters

name
The name of the DSP. This should match the description passed to Studio::System::registerPlugin.
Return Values

If the function succeeds then the return 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

- Studio::System::registerPlugin
Firelight Technologies FMOD Studio API
Studio::System::update

Updates the Studio System. This should be called once per 'game' tick, or once per frame in your application.

C++ Syntax

```cpp
FMOD_RESULT Studio::System::update();
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_System_Update(FMOD_STUDIO_SYSTEM *system);
```

C# Syntax

```csharp
RESULT Studio.System.update();
```

JavaScript Syntax

```javascript
System.update();
```
Return Values

If the function succeeds then the return value 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 Studio has been initialized in asynchronous mode, calling update flips a command buffer for commands to be executed on the async thread and then immediately returns. It is very fast since it is not executing any commands itself. When Studio has been initialized with `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE`, update will execute the queued commands before returning.

If you do not call `Studio::System::update` then previous commands will not be executed. While most of the API hides this behaviour with use of shadowed variables, it can cause unexpected results if waiting in a loop for `Studio::EventDescription::getSampleLoadingState` or `Studio::Bank::getLoadingState` without calling update first.
See Also

- Studio::System::initialize
- Studio::System::flushCommands
Firelight Technologies FMOD Studio API
Studio::EventDescription

The description for a FMOD Studio Event.
Functions

Studio::EventDescription::createInstance
Studio::EventDescription::getID
Studio::EventDescription::getInstanceCount
Studio::EventDescription::getInstanceList
Studio::EventDescription::getLength
Studio::EventDescription::getMaximumDistance
Studio::EventDescription::getMinimumDistance
Studio::EventDescription::getParameter
Studio::EventDescription::getParameterByIndex
Studio::EventDescription::getParameterCount
Studio::EventDescription::getPath
Studio::EventDescription::getSampleLoadingState
Studio::EventDescription::getSoundSize
Studio::EventDescription::getUserData
Studio::EventDescription::getUserProperty
Studio::EventDescription::getUserPropertyByIndex
Studio::EventDescription::getUserPropertyCount
Studio::EventDescription::hasCue
Studio::EventDescription::is3D
Studio::EventDescription::isOneshot
Studio::EventDescription::isSnapshot
Studio::EventDescription::isStream
Studio::EventDescription::loadSampleData
Studio::EventDescription::releaseAllInstances
Studio::EventDescription::setCallback
Studio::EventDescription::setUserData
Studio::EventDescription::unloadSampleData
Remarks

Event Descriptions belong to banks and can be queried after the relevant bank has been loaded. Event Descriptions can be found either by looking up by GUID or by querying all descriptions as part of a bank.
See Also

- Studio::System::loadBankFile
- Studio::System::loadBankMemory
- Studio::System::loadBankCustom
- Studio::System::getEvent
- Studio::Bank::getEventCount
- Studio::Bank::getEventList
Firelight Technologies FMOD Studio API
**Studio::EventDescription::createInstance**

Creates a playable instance of the event / snapshot.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::createInstance(
    Studio::EventInstance **instance);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_CreateInstance(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_STUDIO_EVENTINSTANCE **instance);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.createInstance(
    out EventInstance instance
);
```

**JavaScript Syntax**

```javascript
EventDescription.createInstance(
    instance // writes value to instance.val
);
```
Parameters

instance
    Address of a variable to receive the EventInstance 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 creates playable instances of the EventDescription. Many instances can be created from a single EventDescription. When an event instance is created, it begins loading the required sample data asynchronously; use Studio::EventDescription::getSampleLoadingState to check the loading status.

Sample data can be loaded ahead of time with Studio::EventDescription::loadSampleData or Studio::Bank::loadSampleData.
See Also

- Studio::EventDescription::isSnapshot
- Studio::EventDescription::getSampleLoadingState
- Studio::EventInstance::release

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
Studio::EventDescription::getID

Retrieves the ID of the event.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getID(  
    FMOD_GUID *id 
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetID(  
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,  
    FMOD_GUID *id 
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.getID(  
    out Guid id 
);
```

**JavaScript Syntax**

```javascript
EventDescription.getID(  
    id // writes value to id.val 
);
```
Parameters

id
   Address of a variable to receive the 128-bit GUID.
Return Values

If the function succeeds then the return 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 Studio API
Studio::EventDescription::getInstanceCount

Retrieves the number of created instances for this event type.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getInstanceCount(
    int *count
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetInstanceCount(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    int *count
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.getInstanceCount(
    out int count
);
```

**JavaScript Syntax**

```javascript
EventDescription.getInstanceCount(
    count
); // writes value to count.val
```
Parameters

count
    Address of a variable to receive the number of created instances.
Return Values

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

Used in conjunction with Studio::EventDescription::getInstanceList to track the playing instances of this event.
See Also

- Studio::EventDescription::getInstanceList
Firelight Technologies FMOD Studio API
Studio::EventDescription::getInstanceList

Retrieves the created EventInstances for this event type.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getInstanceList(
    Studio::EventInstance **array,
    int capacity,
    int *count
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetInstanceList(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_STUDIO_EVENTINSTANCE **array,
    int capacity,
    int *count
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.getInstanceList(
    out EventInstance[] array
);
```

**JavaScript Syntax**

```javascript
EventDescription.getInstanceList(
    array, // writes value to array.val
    capacity, // writes value to count.val
    count
);
```
Parameters

array
   An array of memory allocated by the user.
capacity
   The capacity of the array passed in as the first parameter
count
   Address of a variable to receive the number of Event Instances written to 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**

Can be used in conjunction with `Studio::EventDescription::getInstanceCount` to track the playing instances of this event.
See Also

- Studio::EventDescription::getInstanceCount
Firelight Technologies FMOD Studio API
Studio::EventDescription::getLength

Retrieves the length of the event's timeline.

C++ Syntax

FMOD_RESULT Studio::EventDescription::getLength(
    int *length
);

C Syntax

FMOD_RESULT FMOD_Studio_EventDescription_GetLength(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    int *length
);

C# Syntax

RESULT Studio.EventDescription.getLength(
    out int length
);

JavaScript Syntax

EventDescription.getLength(
    length // writes value to length.val
);
Parameters

length
   Address of a variable to receive the timeline length 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

- Studio::EventInstance::getTimelinePosition
- Studio::EventInstance::setTimelinePosition
Firelight Technologies FMOD Studio API
Studio::EventDescription::getMaximumDistance

Retrieves the maximum distance for 3D attenuation of the event.

C++ Syntax

FMOD_RESULT Studio::EventDescription::getMaximumDistance(
    float *distance
);

C Syntax

FMOD_RESULT FMOD_Studio_EventDescription_GetMaximumDistance(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    float *distance
);

C# Syntax

RESULT Studio.EventDescription.getMaximumDistance(
    out float distance
);

JavaScript Syntax

EventDescription.getMaximumDistance(
    distance // writes value to distance.val
);
Parameters

distance
    Address of a variable to receive the distance.


## Return Values

If the function succeeds then the return value 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 largest Max Distance value out of all of the event's 3D Panners, or zero if there are no 3D Panners.
Firelight Technologies FMOD Studio API
Studio::EventDescription::getMinimumDistance

Retrieves the minimum distance for 3D attenuation of the event.

C++ Syntax

FMOD_RESULT Studio::EventDescription::getMinimumDistance(
    float *distance
);

C Syntax

FMOD_RESULT FMOD_Studio_EventDescription_GetMinimumDistance(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    float *distance
);

C# Syntax

RESULT Studio.EventDescription.getMinimumDistance(
    out float distance
);

JavaScript Syntax

EventDescription.getMinimumDistance(
    distance // writes value to distance.val
);
Parameters

distance
  Address of a variable to receive the distance.
Return Values

If the function succeeds then the return value 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 smallest Min Distance value out of all of the event's 3D Panners, or zero if there are no 3D Panners.
Firelight Technologies FMOD Studio API
**Studio::EventDescription::getParameter**

Retrieves an event parameter by name.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getParameter(
    const char *name,
    FMOD_STUDIO_PARAMETER_DESCRIPTION *parameter
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetParameter(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    const char *name,
    FMOD_STUDIO_PARAMETER_DESCRIPTION *parameter
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.getParameter(
    string name,
    out PARAMETER_DESCRIPTION parameter
);
```

**JavaScript Syntax**

```javascript
EventDescription.getParameter(
    name,  
    parameter                   // writes value to parameter.val
);
```
Parameters

name
   Name of the parameter (case-insensitive).
parameter
   Address of a variable to receive the parameter description.
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_STUDIO_PARAMETER_DESCRIPTION](#)
- [Studio::EventInstance::getParameterValue](#)
- [Studio::EventInstance::getParameterValueByIndex](#)
- [Studio::EventInstance::setParameterValue](#)
- [Studio::EventInstance::setParameterValueByIndex](#)
Firelight Technologies FMOD Studio API
Studio::EventDescription::getParameterByIndex

Retrieves an event parameter by index.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getParameterByIndex(
    int index,
    FMOD_STUDIO_PARAMETER_DESCRIPTION *parameter
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetParameterByIndex(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    int index,
    FMOD_STUDIO_PARAMETER_DESCRIPTION *parameter
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.getParameterByIndex(
    int index,
    out PARAMETER_DESCRIPTION parameter
);
```

**JavaScript Syntax**

```javascript
EventDescription.getParameterByIndex(
    index,
    parameter // writes value to parameter.val
);
```
Parameters

index
  Index of the parameter.
parameter
  Address of a variable to receive the parameter 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.
Remarks

This function can be used in combination with `Studio::EventDescription::getParameterCount` to enumerate all parameters within an event.

**NOTE**: The order of parameters is not necessarily the same as what is shown in the FMOD Studio event window.
See Also

- **FMOD_STUDIO_PARAMETER_DESCRIPTION**
- **Studio::EventDescription::getParameterCount**
Firelight Technologies FMOD Studio API
Studio::EventDescription::getParameterCount

Retrieves the number of parameters in the event.

**C++ Syntax**

FMOD_RESULT Studio::EventDescription::getParameterCount(
    int *count
);

**C Syntax**

FMOD_RESULT FMOD_Studio_EventDescription_GetParameterCount(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    int *count
);

**C# Syntax**

RESULT Studio.EventDescription.getParameterCount(
    out int count
);

**JavaScript Syntax**

EventDescription.getParameterCount(
    count // writes value to count.val
);
Parameters

count
   Address of a variable to receive the parameter count.
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

- Studio::EventDescription::getParameterByIndex
Firelight Technologies FMOD Studio API
Studio::EventDescription::getPath

Retrieves the path of the event.

C++ Syntax

FMOD_RESULT Studio::EventDescription::getPath(
    char *path,
    int size,
    int *retrieved
);

C Syntax

FMOD_RESULT FMOD_Studio_EventDescription_GetPath(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    char *path,
    int size,
    int *retrieved
);

C# Syntax

RESULT Studio.EventDescription.getPath(
    out string path
);

JavaScript Syntax

EventDescription.getPath(
    path,
    size,
    retrieved // writes value to retrieved.val
);

// writes value to path.val
Parameters

path
  Address of a buffer to receive the path. Specify 0 or NULL to ignore.

size
  Size of the path buffer in bytes. Required if path parameter is not NULL.

retrieved
  Address of a variable to receive the size of the retrieved path in bytes, including the terminating null character. 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 will return `FMOD_ERR_EVENT_NOTFOUND` unless string data for the requested event is loaded (by loading the "Master Bank.strings.bank" file).

If the retrieved path is too long to fit in the supplied buffer, it will be truncated and this function will return `FMOD_ERR_TRUNCATED`. Use the `retrieved` parameter to obtain the minimum buffer size required to hold the full path.

**JavaScript only:**

Note: For the "path" parameter, the maximum string length is 512.
Firelight Technologies FMOD Studio API
Studio::EventDescription::getSampleLoadingState

Retrieves the sample data loading state of the event.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getSampleLoadingState(
    FMOD_STUDIO_LOADING_STATE *state
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetSampleLoadingState(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_STUDIO_LOADING_STATE *state
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.getSampleLoadingState(
    out LOADING_STATE state
);
```

**JavaScript Syntax**

```javascript
EventDescription.getSampleLoadingState(
    state // writes value to state.val
);
```
Parameters

state
   Address of a variable to receive the loading 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

Event sample data loading can be triggered by calls to
Studio::EventDescription::loadSampleData, Studio::Bank::loadSampleData or
Studio::EventDescription::createInstance.

If the event is invalid, then the state will be set to
FMOD_STUDIO>Loading_State_Unloaded.”
See Also

- `FMOD_STUDIO_LOADING_STATE`
- `Studio::EventDescription::loadSampleData`
- `Studio::Bank::loadSampleData`
Firelight Technologies FMOD Studio API
Studio::EventDescription::getSoundSize

Retrieves the sound size for 3D panning of the event.

C++ Syntax

```cpp
FMOD_RESULT Studio::EventDescription::getSoundSize(
    float *size
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetSoundSize(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    float *size
);
```

C# Syntax

```csharp
RESULT Studio.EventDescription.getSoundSize(
    out float size
);
```

JavaScript Syntax

```javascript
EventDescription.getSoundSize(
    size
    // writes value to size.val
);
```
**Parameters**

`size`

Address of a variable to receive the sound 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.
Remarks

This is the largest Sound Size value out of all of the event's 3D Panners, or zero if there are no 3D Panners.
Firelight Technologies FMOD Studio API
**Studio::EventDescription::getUserData**

Retrieves the user data that is set on the event.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getUserData(
    void **userData
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetUserData(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    void **userData
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.getUserData(
    out IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
EventDescription.getUserData(
    userdata // writes value to userdata.val
);
```
Parameters

userData
   Address of a variable to receive the 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.
See Also

- Studio::EventDescription::setUserData
Firelight Technologies FMOD Studio API
**Studio::EventDescription::getUserProperty**

Retrieves a user property by name.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getUserProperty(
    const char *name,
    FMOD_STUDIO_USER_PROPERTY *property
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetUserProperty(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    const char *name,
    FMOD_STUDIO_USER_PROPERTY *property
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.getUserProperty(
    string name,
    out USER_PROPERTY property
);
```

**JavaScript Syntax**

```javascript
EventDescription.getUserProperty(
    name,
    propertycosity property.val
);
```
**Parameters**

name
   Name of the user property
property
   Address of a variable to receive the user 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

- [FMOD_STUDIO_USER_PROPERTY](#)
- [Studio::EventDescription::getUserPropertyCount](#)
- [Studio::EventDescription::getUserPropertyByIndex](#)
Firelight Technologies FMOD Studio API
Studio::EventDescription::getUserPropertyByIndex

Retrieves a user property by index.

C++ Syntax

FMOD_RESULT Studio::EventDescription::getUserPropertyByIndex(
    int index,
    FMOD_STUDIO_USER_PROPERTY *property
);

C Syntax

FMOD_RESULT FMOD_Studio_EventDescription_GetUserPropertyByIndex(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    int index,
    FMOD_STUDIO_USER_PROPERTY *property
);

C# Syntax

RESULT Studio.EventDescription.getUserPropertyByIndex(
    int index,
    out USER_PROPERTY property
);

JavaScript Syntax

EventDescription.getUserPropertyByIndex(
    index,
    property // writes value to property.val
);
Parameters

index
   Index of the user property.
property
   Address of a variable to receive the user 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

- FMOD_STUDIO_USER_PROPERTY
- Studio::EventDescription::getUserPropertyCount
- Studio::EventDescription::getUserProperty
Firelight Technologies FMOD Studio API
Studio::EventDescription::getUserPropertyCount

Retrieves the number of user properties attached to the event.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::getUserPropertyCount(
    int *count
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_GetUserPropertyCount(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    int *count
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.GetUserPropertyCount(
    out int count
);
```

**JavaScript Syntax**

```javascript
EventDescription.getUserPropertyCount(
    count // writes value to count.val
);
```
Parameters

count
    Address of a variable to receive the user property count.
Return Values

If the function succeeds then the return 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

- Studio::EventDescription::getUserPropertyByIndex
- Studio::EventDescription::getUserProperty
Firelight Technologies FMOD Studio API
Studio::EventDescription::hasCue

Retrieves whether the event has any sustain points.

C++ Syntax

FMOD_RESULT Studio::EventDescription::hasCue(
    bool *cue
);

C Syntax

FMOD_RESULT FMOD_Studio_EventDescription_HasCue(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_BOOL *cue
);

C# Syntax

RESULT Studio.EventDescription.hasCue(
    out bool cue
);

JavaScript Syntax

EventDescription.hasCue(
    cue
    // writes value to cue.val
);
Parameters

cue
   Address of a variable to receive the cue status.
Return Values

If the function succeeds then the return 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

- Studio::EventInstance::triggerCue
Firelight Technologies FMOD Studio API
**Studio::EventDescription::is3D**

Retrieves the event's 3D status, indicating whether the event's behaviour will be affected by its 3D attributes.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::is3D(
    bool *is3D
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_Is3D(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_BOOL *is3D
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.is3D(
    out bool is3D
);
```

**JavaScript Syntax**

```javascript
EventDescription.is3D(
    is3D                                  // writes value to is3D.val
);
```
Parameters

is3D

Address of a variable to receive the 3D status.
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

An event is considered 3D if it contains a 3D panner or an automatic parameter that depends on the event's 3D attributes:

- Distance
- Event Cone Angle
- Event Orientation
- Direction
- Elevation

An event will also return that it is 3D if it has any nested events that are 3D.

A Listener Orientation automatic parameter does not make an event 3D, as it only depends on the listener's 3D attributes.
See Also

- Studio::EventInstance::set3DAttributes
Firelight Technologies FMOD Studio API
Studio::EventDescription::isOneshot

Retrieves the event's oneshot status, indicating whether the event will naturally terminate.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::isOneshot(
    bool *oneshot
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_IsOneshot(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_BOOL *oneshot
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.isOneshot(
    out bool oneshot
);
```

**JavaScript Syntax**

```javascript
EventDescription.isOneshot(
    oneshot // writes value to oneshot.val
);
```
Parameters

oneshot
   Address of a variable to receive the oneshot status.
Return Values

If the function succeeds then the return value 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 event is oneshot then it will naturally terminate and can be used as a fire-and-forget style oneshot sound by calling `Studio::EventInstance::start` followed by `Studio::EventInstance::release`. 
See Also

- Studio::EventInstance::start
- Studio::EventInstance::release
Firelight Technologies FMOD Studio API
**Studio::EventDescription::isSnapshot**

Retrieves whether the event is a snapshot.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::isSnapshot(
    bool *snapshot
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_IsSnapshot(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_BOOL *snapshot
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.isSnapshot(
    out bool snapshot
);
```

**JavaScript Syntax**

```javascript
EventDescription.isSnapshot(
    snapshot // writes value to snapshot.val
);
```
Parameters

snapshot
   Address of a variable to receive the snapshot status.
Return Values

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

Studio implements snapshots as a special kind of event. Use this function to distinguish a snapshot event from a regular event.
Firelight Technologies FMOD Studio API
Studio::EventDescription::isStream

Retrieves the event's stream status, indicating whether the event contains one or more streamed sounds.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::isStream(
    bool *isStream
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_IsStream(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_BOOL *isStream
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.isStream(
    out bool isStream
);
```

**JavaScript Syntax**

```javascript
EventDescription.isStream(
    isStream // writes value to isStream.val
);
```
Parameters

isStream
    Address of a variable to receive the stream status.
Return Values

If the function succeeds then the return 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 Studio API
Studio::EventDescription::loadSampleData

Loads all non-streaming sample data used by the event.

**C++ Syntax**

```
FMOD_RESULT Studio::EventDescription::loadSampleData();
```

**C Syntax**

```
FMOD_RESULT FMOD_Studio_EventDescription_LoadSampleData(FMOD_EVENTDESCRIPTION*);
```

**C# Syntax**

```
RESULT Studio.EventDescription.loadSampleData();
```

**JavaScript Syntax**

```
EventDescription.loadSampleData();
```
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

You can use this function to preload sample data ahead of time so that events can be played immediately when required.

Each time this function is called, it will increment the reference count, so the sample data will not be unloaded until `Studio::EventDescription::unloadSampleData` is called the same number of times.

See `Studio::Bank::loadSampleData` for a description of this function's interaction with `Studio::Bank::loadSampleData` and `Studio::Bank::unloadSampleData`.
See Also

- `Studio::EventDescription::unloadSampleData`
- `Studio::EventDescription::getSampleLoadingState`
- `Studio::Bank::loadSampleData`
- `Studio::Bank::unloadSampleData`
Firelight Technologies FMOD Studio API
Studio::EventDescription::releaseAllInstances

Releases all instances of the event.

C++ Syntax

FMOD_RESULT Studio::EventDescription::releaseAllInstances();

C Syntax

FMOD_RESULT FMOD_Studio_EventDescription_ReleaseAllInstances(FMOD_STUDIO_EVENTDESCRIPTION*);

C# Syntax

RESULT Studio.EventDescription.releaseAllInstances();

JavaScript Syntax

EventDescription.releaseAllInstances();
Return Values

If the function succeeds then the return value 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 immediately stops and releases all instances of the event.
See Also

- Studio::EventInstance::release
Firelight Technologies FMOD Studio API
Studio::EventDescription::setCallback

Sets a default user callback which will be assigned to all future event instances created from the event.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::setCallback(
    FMOD_STUDIO_EVENT_CALLBACK callback,
    FMOD_STUDIO_EVENT_CALLBACK_TYPE callbackmask
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_SetCallback(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    FMOD_STUDIO_EVENT_CALLBACK callback,
    FMOD_STUDIO_EVENT_CALLBACK_TYPE callbackmask
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.setCallback(
    EVENT_CALLBACK callback,
    EVENT_CALLBACK_TYPE callbackmask = EVENT_CALLBACK_TYPE.ALL
);
```

**JavaScript Syntax**

```javascript
EventDescription.setCallback(
    callback,
    callbackmask
);
```
Parameters

callback
   Pointer to a callback function.
callbackmask
   A bitfield specifying which callback types are required. Masking out some
   callback types can help avoid a flood of irrelevant callbacks being
   triggered. Defaults to FMOD_STUDIO_EVENT_CALLBACK_ALL.
Return Values

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

Some callbacks can occur asynchronously depending on Studio initialisation flags. See `Studio::EventInstance::setCallback` for more information about when callbacks occur.

Example:

```cpp
FMOD_RESULT F_CALLBACK MyEventCallback(FMOD_STUDIO_EVENT_CALLBACK_TYPE
FMOD_RESULT SetupEventCallback(Studio::EventDescription *eventDesc)
{
    return eventDesc->setCallback(MyEventCallback, FMOD_STUDIO_EVENT
```

See Also

- Studio::EventInstance::setCallback
- Studio::EventInstance::getPlaybackState
- FMOD_STUDIO_EVENT_CALLBACK
- FMOD_STUDIO_EVENT_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
**Studio::EventDescription::setUserData**

Sets arbitrary user data on the event.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::setUserData(
    void *userData
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_SetUserData(
    FMOD_STUDIO_EVENTDESCRIPTION *eventdescription,
    void *userData
);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
EventDescription.setUserData(
    userdata
);
```
Parameters

userData
    Address of user data to store within the event description 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 (e.g. if many objects of this type exist). Using Studio::EventDescription::getUserData would help in the identification of the object.
See Also

- Studio::EventDescription::getUserData
Firelight Technologies FMOD Studio API
**Studio::EventDescription::unloadSampleData**

Unloads all non-streaming sample data used by the event.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventDescription::unloadSampleData();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventDescription_UnloadSampleData(FMOD_STUDIO_EVENTDESCRIPTION *);
```

**C# Syntax**

```csharp
RESULT Studio.EventDescription.unloadSampleData();
```

**JavaScript Syntax**

```javascript
EventDescription.unloadSampleData();
```
Return Values

If the function succeeds then the return value 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 time this function is called, it will decrement the reference count. If the reference count goes to zero, the sample data will be unloaded.

If any instances of this event exist, sample data will not be unloaded until all instances are released.
See Also

- Studio::EventDescription::loadSampleData
- Studio::EventDescription::getSampleLoadingState
- Studio::Bank::loadSampleData
- Studio::Bank::unloadSampleData
Firelight Technologies FMOD Studio API
Studio::EventInstance

An instance of an FMOD Studio Event.
Functions

Studio::EventInstance::get3DAttributes
Studio::EventInstance::getChannelGroup
Studio::EventInstance::getDescription
Studio::EventInstance::getListenerMask
Studio::EventInstance::getParameter
Studio::EventInstance::getParameterByIndex
Studio::EventInstance::getParameterCount
Studio::EventInstance::getParameterValue
Studio::EventInstance::getParameterValueByIndex
Studio::EventInstance::getPaused
Studio::EventInstance::getPitch
Studio::EventInstance::getPlaybackState
Studio::EventInstance::getProperty
Studio::EventInstance::getReverbLevel
Studio::EventInstance::getTimelinePosition
Studio::EventInstance::getUserData
Studio::EventInstance::getVolume
Studio::EventInstance::isVirtual
Studio::EventInstance::release
Studio::EventInstance::set3DAttributes
Studio::EventInstance::setCallback
Studio::EventInstance::setListenerMask
Studio::EventInstance::setParameterValue
Studio::EventInstance::setParameterValueByIndex
Studio::EventInstance::setParameterValuesByIndices
Studio::EventInstance::setPaused
Studio::EventInstance::setPitch
Studio::EventInstance::setProperty
Studio::EventInstance::setReverbLevel
Studio::EventInstance::setTimelinePosition
Studio::EventInstance::setUserData
Studio::EventInstance::setVolume
Studio::EventInstance::start
Studio::EventInstance::stop
Studio::EventInstance::triggerCue
See Also

- Studio::EventDescription::createInstance
Firelight Technologies FMOD Studio API
Studio::EventInstance::get3DAttributes

Retrieves the 3D position, velocity and orientation of the event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::get3DAttributes(
    FMOD_3D_ATTRIBUTES *attributes
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_Get3DAttributes(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_3D_ATTRIBUTES *attributes
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.get3DAttributes(
    out _3D_ATTRIBUTES attributes
);
```

**JavaScript Syntax**

```javascript
EventInstance.get3DAttributes(
    attributes
    // writes value to attributes.val
);```
Parameters

attributes
  Address of a variable to receive the FMOD_3D_ATTRIBUTES for 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

- FMOD_3D_ATTRIBUTES
- Studio::EventInstance::set3DAttributes
Firelight Technologies FMOD Studio API
Studio::EventInstance::getChannelGroup

Retrieves the Low Level ChannelGroup for the event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::getChannelGroup(
    FMOD::ChannelGroup **group
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetChannelGroup(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_CHANNELGROUP **group
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.getChannelGroup(
    out FMOD.ChannelGroup group
);
```

**JavaScript Syntax**

```javascript
EventInstance.getChannelGroup(
    group // writes value to group.val
);
```
Parameters

group
   Address of a variable to receive 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.
Remarks

The retrieved ChannelGroup corresponds to the master track of the event instance.

It is not possible to get the ChannelGroup until the event instance has finished being created. When Studio has been initialized in asynchronous mode, this function will return **FMOD_ERR_STUDIO_NOT_LOADED** until the **Studio::EventDescription::createInstance** command has been executed in the async thread. When Studio has been initialized with **FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE**, the ChannelGroup will be available after the next **Studio::System::update** call.

To safely obtain the ChannelGroup, you can do one of the following:

- Call **Studio::System::flushCommands** to ensure the instance has finished being created.
- Set up callback to receive **FMOD_STUDIO_EVENT_CALLBACK_CREATED** and obtain the channelgroup there.
- Keep trying to obtain the channelgroup from the instance each frame until it succeeds.
- If running in synchronous mode, call **Studio::System::update** before you get the ChannelGroup.

Also note that when the event instance is destroyed, the underlying ChannelGroup will become invalid. To make sure the ChannelGroup is remains valid, you can do one of the following:

- Do not call **Studio::EventInstance::release** until you have stopped accessing the ChannelGroup.
- After calling **Studio::EventInstance::release**, only access the ChannelGroup from within the event callbacks.
- If running in synchronous mode, the ChannelGroup will become invalid during **Studio::System::update**.
See Also

- Studio::System::initialize
- Studio::System::update
- Studio::System::flushCommands
Firelight Technologies FMOD Studio API
**Studio::EventInstance::getDescription**

Retrieves the EventDescription for the event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::getDescription(
   Studio::EventDescription **description
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetDescription(
   FMOD_STUDIO_EVENTINSTANCE *eventinstance,
   FMOD_STUDIO_EVENTDESCRIPTION **description
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.getDescription(
   out EventDescription description
);
```

**JavaScript Syntax**

```javascript
EventInstance.getDescription(
   description // writes value to description va
);
```
Parameters

description

Address of a variable to receive the EventDescription 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.
Firelight Technologies FMOD Studio API
Studio::EventInstance::getListenerMask

Get the mask of what listeners apply to this event instance.

C++ Syntax

FMOD_RESULT Studio::EventInstance::getListenerMask(
    unsigned int *mask
);

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_GetListenerMask(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    unsigned int *mask
);

C# Syntax

RESULT Studio.EventInstance.getListenerMask(
    out uint mask
);

JavaScript Syntax

EventInstance.getListenerMask(
    mask // writes value to mask.val
);
Parameters

mask
    Address of a variable to receive the mask.
Return Values

If the function succeeds then the return 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

- Studio::EventInstance::setListenerMask
- Studio::System::setNumListeners
Firelight Technologies FMOD Studio API
Studio::EventInstance::getParameter

Retrieves a parameter instance by name.

C++ Syntax

FMOD_RESULT Studio::EventInstance::getParameter(
    const char *name,
    Studio::ParameterInstance **parameter
);

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_GetParameter(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    const char *name,
    FMOD_STUDIO_PARAMETERINSTANCE **parameter
);

C# Syntax

RESULT Studio.EventInstance.getParameter(
    string name,
    out ParameterInstance instance
);

JavaScript Syntax

EventInstance.getParameter(
    name,
    parameter // writes value to parameter.val
);
Parameters

name
   Name of the parameter (case-insensitive).
parameter
   Address of a variable to receive the ParameterInstance 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

**NOTE:** This function is deprecated. It will be removed in a future version. Please get and set parameter values using

- `Studio::EventInstance::getParameterValue`,
- `Studio::EventInstance::setParameterValue`,
- `Studio::EventInstance::getParameterValueByIndex`, and
- `Studio::EventInstance::setParameterValueByIndex`. 
See Also

- Studio::ParameterInstance::setValue
- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::getParameterValueByIndex
- Studio::EventInstance::setParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::EventInstance::getParameterByIndex

Retrieves a parameter instance by index.

C++ Syntax

FMOD_RESULT Studio::EventInstance::getParameterByIndex(
    int index,
    Studio::ParameterInstance **parameter
);

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_GetParameterByIndex(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int index,
    FMOD_STUDIO_PARAMETERINSTANCE **parameter
);

C# Syntax

RESULT Studio.EventInstance.getParameterByIndex(
    int index,
    out ParameterInstance instance
);

JavaScript Syntax

EventInstance.getParameterByIndex(
    index,
    parameter // writes value to parameter.val
);
Parameters

index
    Index of the parameter.
parameter
    Address of a variable to receive the ParameterInstance 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 in combination with 
Studio::EventInstance::getParameterCount to enumerate all parameter instances within an event instance.

NOTE: The order of parameters is not necessarily the same as what is shown in the FMOD Studio event window.

NOTE: This function is deprecated. It will be removed in a future version. Please get and set parameter values using 
Studio::EventInstance::getParameterValue, 
Studio::EventInstance::setParameterValue, 
Studio::EventInstance::getParameterValueByIndex, and 
Studio::EventInstance::setParameterValueByIndex.
See Also

- Studio::EventInstance::getParameterCount
- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::getParameterValueByIndex
- Studio::EventInstance::setParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::EventInstance::getParameterCount

Retrieves the number of parameters in the event instance.

C++ Syntax

FMOD_RESULT Studio::EventInstance::getParameterCount(
    int *count
);

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_GetParameterCount(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int *count
);

C# Syntax

RESULT Studio.EventInstance.getParameterCount(
    out int count
);

JavaScript Syntax

EventInstance.getParameterCount(
    count // writes value to count.val
);
Parameters

count
    Address of a variable to receive the parameter count.
Return Values

If the function succeeds then the return 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

- `Studio::EventInstance::getParameterByIndex`
- `Studio::EventInstance::setParameterByIndex`
Firelight Technologies FMOD Studio API
Studio::EventInstance::getParameterValue

Gets a parameter instance value by name.

C++ Syntax

```cpp
FMOD_RESULT Studio::EventInstance::getParameterValue(
    const char *name,
    float *value,
    float *finalvalue
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetParameterValue(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    const char *name,
    float *value,
    float *finalvalue
);
```

C# Syntax

```csharp
RESULT Studio.EventInstance.getParameterValue(
    string name,
    out float value,
    out float finalvalue
);
```

JavaScript Syntax

```javascript
EventInstance.getParameterValue(
    name, // writes value to value.val
    value, // writes value to finalvalue.val
    finalvalue
);
```
Parameters

name
   Name of the parameter (case-insensitive).
value
   Address of a variable to receive the value as set from the public API. Specify 0 or NULL to ignore.
finalvalue
   Address of a variable to receive the final combined value. 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 finds the parameter by name and then gets the value. This function can return both the user side value set from the last call to `Studio::EventInstance::setParameterValue`, as well as the final combined value which may be the result of automation, modulation, and smoothing. Automatic parameters always return the user value as 0, since they can never be set from the public API.

**NOTE:** The final value will not change immediately. It will change during subsequent asynchronous updates.
See Also

- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::getParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::EventInstance::getParameterValueByIndex

Gets a parameter instance value by index.

C++ Syntax

FMOD_RESULT Studio::EventInstance::getParameterValueByIndex(
    int index,
    float *value,
    float *finalvalue
);

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_GetParameterValueByIndex(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int index,
    float *value,
    float *finalvalue
);

C# Syntax

RESULT Studio.EventInstance.getParameterValueByIndex(
    int index,
    out float value,
    out float finalvalue
);

JavaScript Syntax

EventInstance.getParameterValueByIndex(
    index,
    value,              // writes value to value.val
    finalvalue         // writes value to finalvalue.val
);
Parameters

index
  Index of the parameter.
value
  Address of a variable to receive the value as set from the public API. Specify 0 or NULL to ignore.
finalvalue
  Address of a variable to receive the final combined value. Specify 0 or NULL to ignore.
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 function gets the parameter value for the given index. The parameter index can be one of the list from `Studio::EventInstance::getParameterCount`, or it can found by looking at the index field of `FMOD_STUDIO_PARAMETER_DESCRIPTION` after calling `Studio::EventDescription::getParameter`.

This function can return both the value set from the last call to `Studio::EventInstance::setParameterValue`, as well as the final combined value which may be the result of automation, modulation, and smoothing. Automatic parameters always return the user value as 0, since they can never be set from the public API.

**NOTE**: The final value will not change immediately. It will change during subsequent asynchronous updates.

**NOTE**: The order of parameters is not necessarily the same as what is shown in the FMOD Studio event window.
See Also

- Studio::EventDescription::getParameterCount
- Studio::EventDescription::getParameter
- Studio::EventInstance::getParameterCount
- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::setParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::EventInstance::getPaused

Retrieves the pause state of the event instance.

C++ Syntax

```cpp
FMOD_RESULT Studio::EventInstance::getPaused(
    bool *paused
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetPaused(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_BOOL *paused
);
```

C# Syntax

```csharp
RESULT Studio.EventInstance.getPaused(
    out bool paused
);
```

JavaScript Syntax

```javascript
EventInstance.getPaused(  
    paused // writes value to paused.val
);
```
Parameters

paused
  Address of a variable to receive the pause state.
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

- Studio::EventInstance::setPaused
Studio::EventInstance::getPitch

Retrieves the pitch multiplier set by the API on the event instance.

**C++ Syntax**

```
FMOD_RESULT Studio::EventInstance::getPitch(
    float *pitch,
    float *finalpitch
);
```

**C Syntax**

```
FMOD_RESULT FMOD_Studio_EventInstance_GetPitch(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    float *pitch,
    float *finalpitch
);
```

**C# Syntax**

```
RESULT Studio.EventInstance.getPitch(
    out float pitch,
    out float finalpitch
);
```

**JavaScript Syntax**

```
EventInstance.getPitch(
    pitch,  // writes value to pitch.val
    finalpitch  // writes value to finalpitch.val
);
```
**Parameters**

pitch
   Address of a variable to receive the pitch as set from the public API. Specify 0 or NULL to ignore.

finalpitch
   Address of a variable to receive the final combined pitch. 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 can return both the pitch set from the last call to `Studio::EventInstance::setPitch`, as well as the final combined pitch which may be the result of automation and modulation.

**NOTE:** The final pitch will not change immediately. It will change during subsequent asynchronous updates.
See Also

- Studio::EventInstance::setPitch
Firelight Technologies FMOD Studio API
Studio::EventInstance::getPlaybackState

Retrieves the playback state of the event instance.

C++ Syntax

```cpp
FMOD_RESULT Studio::EventInstance::getPlaybackState(
    FMOD_STUDIO_PLAYBACK_STATE *state
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetPlaybackState(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_STUDIO_PLAYBACK_STATE *state
);
```

C# Syntax

```csharp
RESULT Studio.EventInstance.getPlaybackState(
    out PLAYBACK_STATE state
);
```

JavaScript Syntax

```javascript
EventInstance.getPlaybackState(
    state // writes value to state.val
);
```
Parameters

state
Address of a variable to receive the current playback state of the event instance. See `FMOD_STUDIO_PLAYBACK_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

You can poll this function to track the playback state of an event instance.

If the instance is invalid, then the state will be set to
FMOD_STUDIO_PLAYBACK_STOPPED.
See Also

- Studio::EventInstance::start
- Studio::EventInstance::stop
- FMOD_STUDIO_EVENT_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
**Studio::EventInstance::getProperty**

Retrieves the value of a built-in event instance property.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::getProperty(
    FMOD_STUDIO_EVENT_PROPERTY index,
    float *value
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetProperty(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_STUDIO_EVENT_PROPERTY index,
    float *value
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.getProperty(
    EVENT_PROPERTY index,
    out float value
);
```

**JavaScript Syntax**

```javascript
EventInstance.getProperty(
    index,
    value // writes value to value.val
);
```
Parameters

index
   The index of the property to retrieve.
value
   Address of a variable to receive the 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.
See Also

- [FMOD_STUDIO_EVENT_PROPERTY](#)
- [Studio::EventInstance::setProperty](#)

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
Studio::EventInstance::getReverbLevel

Retrieves the send level to a Low Level reverb instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::getReverbLevel(
    int index,
    float *level
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetReverbLevel(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int index,
    float *level
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.getReverbLevel(
    int index,
    out float level
);
```

**JavaScript Syntax**

```javascript
EventInstance.getReverbLevel(
    index,
    level // writes value to level.val
);
```
Parameters

index
   Index of the Low Level reverb instance to target, from 0 to 3.
level
   Address of a variable to receive the send level for the signal to the reverb, from 0 (none) to 1 (full).
Return Values

If the function succeeds then the return value 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 gets the level of signal sent from the event instance to a Low Level reverb instance.

**NOTE:** Use this function in preference to obtaining the event's ChannelGroup via `Studio::EventInstance::getChannelGroup` and calling `ChannelGroup::getReverbProperties` directly. This function is safe to use anytime.
See Also

- Studio::EventInstance::setReverbLevel
Firelight Technologies FMOD Studio API
**Studio::EventInstance::getTimelinePosition**

Retrieves the position of the event instance's timeline playback cursor.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::getTimelinePosition(
    int *position
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetTimelinePosition(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int *position
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.getTimelinePosition(
    out int position
);
```

**JavaScript Syntax**

```javascript
EventInstance.getTimelinePosition(
    position // writes value to position.val
);
```
Parameters

position
  Address of a variable to receive the timeline position 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.
Firelight Technologies FMOD Studio API
**Studio::EventInstance::getUserData**

Retrieves the user data that is set on the event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::getUserData(
    void **userData
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetUserData(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    void **userData
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.getUserData(
    out IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
EventInstance.getUserData(
    userdata
) // writes value to userdata.val
```
Parameters

userData
Address of a variable to receive the 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.
See Also

- Studio::EventInstance::setUserData
Firelight Technologies FMOD Studio API
**Studio::EventInstance::getVolume**

Retrieves the volume level of the event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::getVolume(
    float *volume,
    float *finalvolume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_GetVolume(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    float *volume,
    float *finalvolume
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.getVolume(
    out float volume,
    out float finalvolume
);
```

**JavaScript Syntax**

```javascript
EventInstance.getVolume(
    volume, // writes value to volume.val
    finalvolume // writes value to finalvolume.val
);
```
Parameters

volume
    Address of a variable to receive the volume as set from the public API. Specify 0 or NULL to ignore.
finalvolume
    Address of a variable to receive the final combined volume. 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 can return both the volume set from the last call to Studio::EventInstance::setVolume, as well as the final combined volume which may be the result of automation and modulation.

**NOTE:** The final volume will not change immediately. It will change during subsequent asynchronous updates.
See Also

- Studio::EventInstance::setVolume
Firelight Technologies FMOD Studio API
**Studio::EventInstance::isVirtual**

Retrieves the virtualization state of the event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::isVirtual(
    bool *virtualState
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_IsVirtual(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_BOOL *virtualState
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.isVirtual(
    out bool virtualState
);
```

**JavaScript Syntax**

```javascript
EventInstance.isVirtual(
    virtualState // writes value to virtualState.v
);
```
Parameters

virtualState
    Address of a variable to receive the virtualization 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

This can be used to check if an event instance has been virtualized due to the max polyphony being exceeded.
Firelight Technologies FMOD Studio API
**Studio::EventInstance::release**

Schedules the event instance to be destroyed when it stops.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::release();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_Release(FMOD_STUDIO_EVENTINSTANCE *);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.release();
```

**JavaScript Syntax**

```javascript
EventInstance.release();
```
Return Values

If the function succeeds then the return value 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 instance is already stopped when release is called, it will be destroyed after the next update.

The handle will remain valid until the event instance stops and is destroyed. Oneshot events will stop naturally, while looping events need to be stopped manually (e.g. by a call to Studio::EventInstance::stop).
See Also

- Studio::EventInstance::stop
Firelight Technologies FMOD Studio API
**Studio::EventInstance::set3DAttributes**

Sets the 3D position, velocity and orientation for the event instance.

**C++ Syntax**

```c++
FMOD_RESULT Studio::EventInstance::set3DAttributes(
    FMOD_3D_ATTRIBUTES *attributes
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_Set3DAttributes(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_3D_ATTRIBUTES *attributes
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.set3DAttributes(
    _3D_ATTRIBUTES attributes
);
```

**JavaScript Syntax**

```javascript
EventInstance.set3DAttributes(
    attributes
);
```
Parameters

attributes
   The 3D attributes to set.
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 function is used to position the event inside the 3D game world. This will update the 3D panner and any automatic Distance or Angle parameters.
See Also

- FMOD_3D_ATTRIBUTES
- Studio::EventInstance::get3DAttributes
- Studio::EventDescription::is3D
Firelight Technologies FMOD Studio API
Studio::EventInstance::setCallback

Sets a user callback for the event instance.

**C++ Syntax**

```c++
FMOD_RESULT Studio::EventInstance::setCallback(
    FMOD_STUDIO_EVENT_CALLBACK callback,
    FMOD_STUDIO_EVENT_CALLBACK_TYPE callbackmask
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetCallback(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_STUDIO_EVENT_CALLBACK callback,
    FMOD_STUDIO_EVENT_CALLBACK_TYPE callbackmask
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.setCallback(
    EVENT_CALLBACK callback,
    EVENT_CALLBACK_TYPE callbackmask = EVENT_CALLBACK_TYPE.ALL
);
```

**JavaScript Syntax**

```javascript
EventInstance.setCallback(
    callback,
    callbackmask
);
```
Parameters

callback
  Pointer to a callback function.

callbackmask
  A bitfield specifying which callback types are required. Masking out some callback types can help avoid a flood of irrelevant callbacks being triggered. Defaults to `FMOD_STUDIO_EVENT_CALLBACK_ALL`. 
Return Values

If the function succeeds then the return value 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 Studio has been initialized in asynchronous mode, callbacks will be fired from the Studio asynchronous thread.

If `FMOD_STUDIO_INIT_DEFERRED_CALLBACKS` is used, then `FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_MARKER` and `FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_BEAT` will be deferred until the next update in the main thread, but the other types will still be called asynchronously.

When Studio has been initialized with `FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE`, all callbacks will be fired from within the update in the main thread.

Example:

```cpp
FMOD_RESULT F_CALLBACK MyEventCallback(FMOD_STUDIO_EVENT_CALLBACK_TYPE type)
{
    return eventInst->setCallback(MyEventCallback, FMOD_STUDIO_EVENT
```
See Also

- Studio::EventDescription::setCallback
- Studio::EventInstance::getPlaybackState
- FMOD_STUDIO_EVENT_CALLBACK
- FMOD_STUDIO_EVENT_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
**Studio::EventInstance::setListenerMask**

Set the mask of what listeners apply to this event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::setListenerMask(
    unsigned int mask
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetListenerMask(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    unsigned int mask
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.setListenerMask(
    uint mask
);
```

**JavaScript Syntax**

```javascript
EventInstance.setListenerMask(
    mask
);
```
Parameters

mask
   Mask of listeners that apply to this event instance.
Return Values

If the function succeeds then the return value 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 default mask is 0xffffffff, which is for all listeners to be enabled.

**NOTE:** It is an error to set a mask that does not include at least one active listener.
See Also

- Studio::EventInstance::getListenerMask
- Studio::System::setNumListeners
Firelight Technologies FMOD Studio API
Studio::EventInstance::setParameterValue

Sets a parameter instance value by name.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::setParameterValue(
    const char *name,
    float value
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetParameterValue(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    const char *name,
    float value
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.setParameterValue(
    string name,
    float value
);
```

**JavaScript Syntax**

```javascript
EventInstance.setParameterValue(
    name,
    value
);
```
Parameters

name
   Name of the parameter (case-insensitive).
value
   Value to set.
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 function finds the parameter by name and then sets the value.

If the event has no parameter of that name, \texttt{FMOD\_ERR\_EVENT\_NOTFOUND} is returned. If it is an automatic parameter, \texttt{FMOD\_ERR\_INVALID\_PARAM} is returned.
See Also

- [Studio::EventInstance::getParameterValue](#)
- [Studio::EventInstance::getParameterValueByIndex](#)
- [Studio::EventInstance::setParameterValueByIndex](#)
Firelight Technologies FMOD Studio API
Studio::EventInstance::setParameterValueByIndex

Sets a parameter instance value by index.

C++ Syntax

FMOD_RESULT Studio::EventInstance::setParameterValueByIndex(
    int index,
    float value
);

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_SetParameterValueByIndex(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int index,
    float value
);

C# Syntax

RESULT Studio.EventInstance.setParameterValueByIndex(
    int index,
    float value
);

JavaScript Syntax

EventInstance.setParameterValueByIndex(
    index,
    value
);
Parameters

index
   Index of the parameter.
value
   Value 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 function sets the parameter value for the given index. The parameter index can be one of the list from Studio::EventInstance::getParameterCount, or it can found by looking at the index field of FMOD_STUDIO_PARAMETER_DESCRIPTION after calling Studio::EventDescription::getParameter.

If it is an automatic parameter, FMOD_ERR_INVALID_PARAM is returned.

NOTE: The order of parameters is not necessarily the same as what is shown in the FMOD Studio event window.
See Also

- Studio::EventInstance::getParameterCount
- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::getParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::EventInstance::setParameterValuesByIndices

Sets multiple parameter instance values by index.

C++ Syntax

```cpp
FMOD_RESULT Studio::EventInstance::setParameterValuesByIndices(
    int *indices,
    float *values,
    int count
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetParameterValuesByIndices(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int *indices,
    float *values,
    int count
);
```

C# Syntax

```csharp
RESULT Studio.EventInstance.setParameterValuesByIndices(
    int[] indices,
    float[] values,
    int count
);
```
**Parameters**

- **indices**
  - Indices of the parameters.
- **values**
  - Values to set.
- **count**
  - Number of indices and values.
Return Values

If the function succeeds then the return value 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 sets the parameter values for the given indices. Each parameter index can be one of the list from `Studio::EventInstance::getParameterCount`, or it can found by looking at the index field of `FMOD_STUDIO_PARAMETER_DESCRIPTION` after calling `Studio::EventDescription::getParameter`. If any index is an automatic parameter, `FMOD_ERR_INVALID_PARAM` is returned without setting the value of any parameters.

If any index is set to -1 then the index and corresponding value will be ignored.

A maximum of 92 parameter values can be set at once. If count is greater than 92, `FMOD_ERR_INVALID_PARAM` is returned.

**NOTE**: The order of parameters is not necessarily the same as what is shown in the FMOD Studio event window.
See Also

- Studio::EventInstance::getParameterCount
- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::setParameterValueByIndex
- Studio::EventInstance::getParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::EventInstance::setPaused

Sets the pause state of the event instance.

C++ Syntax

FMOD_RESULT Studio::EventInstance::setPaused(
    bool paused
);

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_SetPaused(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_BOOL paused
);

C# Syntax

RESULT Studio.EventInstance.setPaused(
    bool paused
);

JavaScript Syntax

EventInstance.setPaused(
    paused
);
**Parameters**

`paused`

The desired pause state. true = pause, 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.
Remarks

This function allows pausing/unpausing of an event instance.
See Also

- Studio::EventInstance::getPaused
Firelight Technologies FMOD Studio API
Studio::EventInstance::setPitch

Sets the pitch multiplier for the event instance.

C++ Syntax

```cpp
FMOD_RESULT Studio::EventInstance::setPitch(
    float pitch
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetPitch(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    float pitch
);
```

C# Syntax

```csharp
RESULT Studio.EventInstance.setPitch(
    float pitch
);
```

JavaScript Syntax

```javascript
EventInstance.setPitch(
    pitch
);
```
Parameters

pitch
   The pitch multiplier. 1 = normal pitch.
Return Values

If the function succeeds then the return value 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 pitch is applied as a multiplier for the event pitch. It does not override the pitch set in FMOD Studio, nor any internal pitch automation or modulation.
See Also

- Studio::EventInstance::getPitch
Firelight Technologies FMOD Studio API
Studio::EventInstance::setProperty

Sets the value of a built-in event instance property.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::setProperty(
    FMOD_STUDIO_EVENTPROPERTY index,
    float value
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetProperty(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_STUDIO_EVENT_PROPERTY index,
    float value
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.setProperty(
    EVENT_PROPERTY index,
    float value
);
```

**JavaScript Syntax**

```javascript
EventInstance.setProperty(
    index,
    value
);
```
Parameters

index
    The index of the property to set.
value
    The property value 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

- **FMOD_STUDIO_EVENTPROPERTY**
- **Studio::EventInstance::getProperty**
Firelight Technologies FMOD Studio API
Studio::EventInstance::setReverbLevel

Sets the send level to a Low Level reverb instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::setReverbLevel(
    int index,
    float level
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetReverbLevel(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int index,
    float level
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.setReverbLevel(
    int index,
    float level
);
```

**JavaScript Syntax**

```javascript
EventInstance.setReverbLevel(
    index,
    level
);
```
Parameters

index
   Index of the Low Level reverb instance to target, from 0 to 3.
level
   Send level for the signal to the reverb, from 0 (none) to 1 (full). 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

This function controls the signal to be sent from the event instance to a Low Level reverb instance.

**NOTE:** Use this function in preference to obtaining the event's ChannelGroup via `Studio::EventInstance::getChannelGroup` and calling `ChannelGroup::setReverbProperties` directly. This function is safe to use anytime.
See Also

- **Studio::EventInstance::getReverbLevel**
- **ChannelGroup::setReverbProperties**
- **System::setReverbProperties**
Firelight Technologies FMOD Studio API
**Studio::EventInstance::setTimelinePosition**

Sets the position of the event instance's timeline playback cursor.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::setTimelinePosition(
    int position
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetTimelinePosition(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    int position
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.setTimelinePosition(
    int position
);
```

**JavaScript Syntax**

```javascript
EventInstance.setTimelinePosition(
    position
);
```
Parameters

position
  Desired timeline position 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.
Remarks

You can use this function to seek the timeline.
Firelight Technologies FMOD Studio API
Studio::EventInstance::setUserData

Sets arbitrary user data on the event instance.

C++ Syntax

```cpp
FMOD_RESULT Studio::EventInstance::setUserData(
    void *userData
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetUserData(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    void *userData
);
```

C# Syntax

```csharp
RESULT Studio.EventInstance.setUserData(
    IntPtr userdata
);
```

JavaScript Syntax

```javascript
EventInstance.setUserData(
    userdata
);
```
Parameters

userData
   Address of user data to be stored within the event instance 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

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 `Studio::EventInstance::getUserData` would help in the identification of the object.
See Also

- Studio::EventInstance::getUserData
Firelight Technologies FMOD Studio API
Studio::EventInstance::setVolume

Sets the volume level of the event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::setVolume(
    float volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_SetVolume(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    float volume
);
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.setVolume(
    float volume
);
```

**JavaScript Syntax**

```javascript
EventInstance.setVolume(  
    volume
);
```
Parameters

volume
The volume as a linear gain. 0 = silent, 1 = full volume.
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 volume is applied as a scaling factor for the event volume. It does not override the volume level set in FMOD Studio, nor any internal volume automation or modulation.
See Also

- **Studio::EventInstance::getVolume**
Firelight Technologies FMOD Studio API
**Studio::EventInstance::start**

Starts replay of the event instance.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::EventInstance::start();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_EventInstance_Start(FMOD_STUDIO_EVENTINSTANCE *
```

**C# Syntax**

```csharp
RESULT Studio.EventInstance.start();
```

**JavaScript Syntax**

```javascript
EventInstance.start();
```
Return Values

If the function succeeds then the return value 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 begin replay of the event instance. If the instance was already playing it will restart the event.
See Also

- Studio::EventInstance::stop
Firelight Technologies FMOD Studio API
Studio::EventInstance::stop

Stops playback of the event instance.

C++ Syntax

FMOD_RESULT Studio::EventInstance::stop(
    FMOD_STUDIO_STOP_MODE mode
);

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_Stop(
    FMOD_STUDIO_EVENTINSTANCE *eventinstance,
    FMOD_STUDIO_STOP_MODE mode
);

C# Syntax

RESULT Studio.EventInstance.stop(
    STOP_MODE mode
);

JavaScript Syntax

EventInstance.stop(
    mode
);
Parameters

mode
   The desired stop mode. See FMOD_STUDIO_STOP_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

- FMOD_STUDIO_STOP_MODE
- Studio::EventInstance::start
Firelight Technologies FMOD Studio API
Studio::EventInstance::triggerCue

Triggers a cue on the event, which allows the timeline cursor to move past sustain points.

C++ Syntax

FMOD_RESULT Studio::EventInstance::triggerCue();

C Syntax

FMOD_RESULT FMOD_Studio_EventInstance_TriggerCue(FMOD_STUDIO_EVENT_INSTANCE);

C# Syntax

RESULT Studio.EventInstance.triggerCue();

JavaScript Syntax

EventInstance.triggerCue();
**Return Values**

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

Triggering cues makes the timeline cursor continue past sustain points. The cue can be triggered ahead of time; for each time it is triggered, the timeline cursor will continue past one more sustain point.

This function returns **FMOD_ERR_EVENT_NOTFOUND** if the event has no sustain points.
See Also

- Studio::EventDescription::hasCue
Firelight Technologies FMOD Studio API
Studio::ParameterInstance

A parameter instance for an FMOD Studio Event.
Functions

Studio::ParameterInstance::getDescription Studio::ParameterInstance::getValue
Studio::ParameterInstance::setValue
Remarks

**NOTE**: This class is deprecated. It will be removed in a future version. Please get and set parameter values using `Studio::EventInstance::getParameterValue`, `Studio::EventInstance::setParameterValue`, `Studio::EventInstance::getParameterValueByIndex`, and `Studio::EventInstance::setParameterValueByIndex`. 
See Also

- Studio::EventInstance::getParameter
- Studio::EventInstance::getParameterByIndex
- Studio::EventInstance::getParameterCount
- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::setParameterValueByIndex
- Studio::EventInstance::setParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::ParameterInstance::getDescription

Retrieves the description for the parameter.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::ParameterInstance::getDescription(
    FMOD_STUDIO_PARAMETER_DESCRIPTION *description
);```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_ParameterInstance_GetDescription(
    FMOD_STUDIO_PARAMETERINSTANCE *parameterinstance,
    FMOD_STUDIO_PARAMETER_DESCRIPTION *description
);```

**C# Syntax**

```csharp
RESULT Studio.ParameterInstance.getDescription(
    out PARAMETER_DESCRIPTION description
);```

**JavaScript Syntax**

```javascript
ParameterInstance.getDescription(
    description // writes value to description.va
);```
Parameters

description
    Address of a variable to receive the parameter 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.
Remarks

**NOTE:** This class is deprecated. It will be removed in a future version. Please get and set parameter values using `Studio::EventInstance::getParameterValue`, `Studio::EventInstance::setParameterValue`, `Studio::EventInstance::getParameterValueByIndex`, and `Studio::EventInstance::setParameterValueByIndex`. 
See Also

- **FMOD_STUDIO_PARAMETER_DESCRIPTION**
- **Studio::EventInstance::getParameterValue**
- **Studio::EventInstance::setParameterValue**
- **Studio::EventInstance::getParameterValueByIndex**
- **Studio::EventInstance::setParameterValueByIndex**
Firelight Technologies FMOD Studio API
Studio::ParameterInstance::getValue

Retrieves the value of the parameter.

C++ Syntax

FMOD_RESULT Studio::ParameterInstance::getValue(
    float *value
);

C Syntax

FMOD_RESULT FMOD_Studio_ParameterInstance_GetValue(
    FMOD_STUDIO_PARAMETERINSTANCE *parameterinstance,
    float *value
);

C# Syntax

RESULT Studio.ParameterInstance.getValue(
    out float value
);

JavaScript Syntax

ParameterInstance.getValue(
    value // writes value to value.val
);
Parameters

value
   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.
Remarks

NOTE: This class is deprecated. It will be removed in a future version. Please get and set parameter values using `Studio::EventInstance::getParameterValue`, `Studio::EventInstance::setParameterValue`, `Studio::EventInstance::getParameterValueByIndex`, and `Studio::EventInstance::setParameterValueByIndex`. 
See Also

- Studio::ParameterInstance::setValue
- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::getParameterValueByIndex
- Studio::EventInstance::setParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::ParameterInstance::setValue

Sets the value of the parameter.

C++ Syntax

FMOD_RESULT Studio::ParameterInstance::setValue(
    float value
);

C Syntax

FMOD_RESULT FMOD_Studio_ParameterInstance_SetValue(
    FMOD_STUDIO_PARAMETERINSTANCE *parameterinstance,
    float value
);

C# Syntax

RESULT Studio.ParameterInstance.setValue(
    float value
);

JavaScript Syntax

ParameterInstance.setValue(
    value
);
Parameters

value
   The value 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

Parameters are used to control event playback and link it to game state. The provided value will be clamped to the parameter's range before it is set.

**NOTE:** This class is deprecated. It will be removed in a future version. Please get and set parameter values using `Studio::EventInstance::getParameterValue`, `Studio::EventInstance::setParameterValue`, `Studio::EventInstance::getParameterValueByIndex`, and `Studio::EventInstance::setParameterValueByIndex`. 
See Also

- Studio::ParameterInstance::getValue
- Studio::EventInstance::getParameterValue
- Studio::EventInstance::setParameterValue
- Studio::EventInstance::getParameterValueByIndex
- Studio::EventInstance::setParameterValueByIndex
Firelight Technologies FMOD Studio API
Studio::Bus

Represents a global mixer bus.
Functions

`Studio::Bus::getChannelGroup`  `Studio::Bus::getID`
`Studio::Bus::getMute`
`Studio::Bus::getPath`
`Studio::Bus::getPaused`
`Studio::Bus::getVolume`
`Studio::Bus::lockChannelGroup`
`Studio::Bus::setMute`
`Studio::Bus::setPaused`
`Studio::Bus::setVolume`
`Studio::Bus::stopAllEvents`
`Studio::Bus::unlockChannelGroup`
See Also

- [Studio::System::getBus](#)
Firelight Technologies FMOD Studio API
Studio::Bus::getChannelGroup

Retrieves the Low Level ChannelGroup used by the bus.

C++ Syntax

FMOD_RESULT Studio::Bus::getChannelGroup(
    FMOD::ChannelGroup **group
);

C Syntax

FMOD_RESULT FMOD_Studio_Bus_GetChannelGroup(
    FMOD_STUDIO_BUS *bus,
    FMOD_CHANNELGROUP **group
);

C# Syntax

RESULT Studio.Bus.getChannelGroup(
    out FMOD.ChannelGroup group
);

JavaScript Syntax

Bus.getChannelGroup(  
    channelgroup
      // writes value to channelgroup.v
);
Parameters

group
    Address of a variable to receive a pointer to the Low Level ChannelGroup.
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

The ChannelGroup is created and destroyed on demand. This means it only exists if at least one event instance routes into the bus. If it doesn't exist, this function will return **FMOD_ERR_STUDIO_NOT_LOADED**.

You can force the ChannelGroup to be created by calling **Studio::Bus::lockChannelGroup.**
See Also

- Studio::Bus::lockChannelGroup
- Studio::Bus::unlockChannelGroup
Firelight Technologies FMOD Studio API
Studio::Bus::getID

Retrieves the ID of the bus.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bus::getID(
    FMOD_GUID *id
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bus_GetID(
    FMOD_STUDIO_BUS *bus,
    FMOD_GUID *id
);
```

**C# Syntax**

```csharp
RESULT Studio.Bus.getID(
    out Guid id
);
```

**JavaScript Syntax**

```javascript
Bus.getID(
    id
) // writes value to id.val
```
**Parameters**

*id*

Address of a variable to receive the 128-bit GUID.
Return Values

If the function succeeds then the return 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 Studio API
**Studio::Bus::getMute**

Retrieves the mute state of the bus.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bus::getMute(
    bool *mute
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bus_GetMute(
    FMOD_STUDIO_BUS *bus,
    FMOD_BOOL *mute
);
```

**C# Syntax**

```csharp
RESULT Studio.Bus.getMute(
    out bool mute
);
```

**JavaScript Syntax**

```javascript
Bus.getMute(
    mute
    // writes value to mute.val
);
```
Parameters

mute
    Address of a variable to receive the mute 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

- Studio::Bus::setMute
Firelight Technologies FMOD Studio API
Studio::Bus::getPath

Retrieves the path of the bus.

C++ Syntax

FMOD_RESULT Studio::Bus::getPath(
    char *path,
    int size,
    int *retrieved
);

C Syntax

FMOD_RESULT FMOD_Studio_Bus_GetPath(
    FMOD_STUDIO_BUS *bus,
    char *path,
    int size,
    int *retrieved
);

C# Syntax

RESULT Studio.Bus.getPath(
    out string path
);

JavaScript Syntax

Bus.getPath(
    path,              // writes value to path.val
    size,             // writes value to retrieved.val
    retrieved
);
Parameters

path
Address of a buffer to receive the path. Specify 0 or NULL to ignore.

size
Size of the path buffer in bytes. Required if path parameter is not NULL.

retrieved
Address of a variable to receive the size of the retrieved path in bytes, including the terminating null character. 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 will return `FMOD_ERR_EVENT_NOTFOUND` unless string data for the requested bus is loaded (by loading the "Master Bank.strings.bank" file).

If the retrieved path is too long to fit in the supplied buffer, it will be truncated and this function will return `FMOD_ERR_TRUNCATED`. Use the `retrieved` parameter to obtain the minimum buffer size required to hold the full path.

**JavaScript only:**

Note: For the "path" parameter, the maximum string length is 512.
Firelight Technologies FMOD Studio API
**Studio::Bus::getPaused**

Retrieves the pause state of the bus.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bus::getPaused(
    bool *paused
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bus_GetPaused(
    FMOD_STUDIO_BUS *bus,
    FMOD_BOOL *paused
);
```

**C# Syntax**

```csharp
RESULT Studio.Bus.getPaused(
    out bool paused
);
```

**JavaScript Syntax**

```javascript
Bus.getPaused(
    paused // writes value to paused.val
);
```
Parameters

paused
   Address of a variable to receive the pause state.
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

- Studio::Bus::setPaused
Firelight Technologies FMOD Studio API
Studio::Bus::getVolume

Retrieves the volume level of the bus.

C++ Syntax

FMOD_RESULT Studio::Bus::getVolume(
    float *volume,
    float *finalvolume
);

C Syntax

FMOD_RESULT FMOD_Studio_Bus_GetVolume(
    FMOD_STUDIO_BUS *bus,
    float *volume,
    float *finalvolume
);

C# Syntax

RESULT Studio.Bus.getVolume(
    out float volume,
    out float finalvolume
);

JavaScript Syntax

Bus.getVolume(
    volume, // writes value to volume.val
    finalvolume // writes value to finalvolume.val
);
Parameters

volume
Address of a variable to receive the volume as set from the public API. Specify 0 or NULL to ignore.

finalvolume
Address of a variable to receive the final combined volume. 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 can return both the volume set from the last call to Studio::Bus::setVolume, as well as the final combined volume which may be the result of automation and modulation.

**NOTE:** The final volume will not change immediately. It will change during subsequent asynchronous updates.
See Also

- Studio::Bus::setVolume
Firelight Technologies FMOD Studio API
Studio::Bus::lockChannelGroup

Locks the Low Level ChannelGroup used by the bus.

C++ Syntax

FMOD_RESULT Studio::Bus::lockChannelGroup();

C Syntax

FMOD_RESULT FMOD_Studio_Bus_LockChannelGroup(FMOD_STUDIO_BUS *bus);

C# Syntax

RESULT Studio.Bus.lockChannelGroup();

JavaScript Syntax

Bus.lockChannelGroup();
Return Values

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

Normally the ChannelGroup is created and destroyed on demand. This means it only exists if at least one event instance routes into the bus. This function forces the ChannelGroup to be created and to persist until Studio::Bus::unlockChannelGroup is called.

Note that the ChannelGroup may not be available immediately after calling this function. When Studio has been initialized in asynchronous mode, the ChannelGroup will not be created until the Studio::Bus::lockChannelGroup command has been executed in the async thread. When Studio has been initialized with FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE, the ChannelGroup will be available after the next Studio::System::update call.

You can call Studio::System::flushCommands to ensure the ChannelGroup has been created. Alternatively you can keep trying to obtain the ChannelGroup with Studio::Bus::getChannelGroup until it is ready.
See Also

- `Studio::Bus::unlockChannelGroup`
- `Studio::Bus::getChannelGroup`
- `Studio::System::initialize`
- `Studio::System::update`
- `Studio::System::flushCommands`
Firelight Technologies FMOD Studio API
Studio::Bus::setMute

Sets the mute state of the bus.

C++ Syntax

FMOD_RESULT Studio::Bus::setMute(
   bool mute
);

C Syntax

FMOD_RESULT FMOD_Studio_Bus_SetMute(  
   FMOD_STUDIO_BUS *bus,  
   FMOD_BOOL mute
);

C# Syntax

RESULT Studio.Bus.setMute(  
   bool mute
);

JavaScript Syntax

Bus.setMute(  
   mute
);
Parameters

mute

The desired mute state. true = mute, false = unmute.
Return Values

If the function succeeds then the return value 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 allows muting/unmuting of all audio routed into the bus. If mute is true, it overrides the mute state of all objects routed into this bus. If mute is false, objects routed into this bus obey their own mute state.
See Also

- Studio::Bus::getMute
Firelight Technologies FMOD Studio API
Studio::Bus::setPaused

Sets the pause state of the bus.

C++ Syntax

FMOD_RESULT Studio::Bus::setPaused(
   bool paused
);

C Syntax

FMOD_RESULT FMOD_Studio_Bus_SetPaused(
   FMOD_STUDIO_BUS *bus,
   FMOD_BOOL paused
);

C# Syntax

RESULT Studio.Bus.setPaused(
   bool paused
);

JavaScript Syntax

Bus.setPaused(
   paused
);
Parameters

paused
   The desired pause state. true = pause, 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.
Remarks

This function allows pausing/unpausing of all audio routed into the bus. If paused is true, it overrides the pause state of all objects routed into this bus. If paused is false, objects routed into this bus obey their own pause state.
See Also

- `Studio::Bus::getPaused`
Firelight Technologies FMOD Studio API
Studio::Bus::setVolume

Sets the volume level of the bus.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bus::setVolume(
    float volume
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bus_SetVolume(
    FMOD_STUDIO_BUS *bus,
    float volume
);
```

**C# Syntax**

```csharp
RESULT Studio.Bus.setVolume(
    float volume
);
```

**JavaScript Syntax**

```javascript
Bus.setVolume(
    volume
);
```
Parameters

volume
   The volume level to set as a linear gain. 0 = silent, 1 = 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 allows volume control of all audio routed into the bus. This fader level is relative to the fader level set in FMOD Studio.
See Also

- Studio::Bus::getVolume
Firelight Technologies FMOD Studio API
**Studio::Bus::stopAllEvents**

Stops all EventInstances routed into the bus.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bus::stopAllEvents(
    FMOD_STUDIO_STOP_MODE mode
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bus_StopAllEvents(
    FMOD_STUDIO_BUS *bus,
    FMOD_STUDIO_STOP_MODE mode
);
```

**C# Syntax**

```csharp
RESULT Studio.Bus.stopAllEvents(
    STOP_MODE mode
);
```

**JavaScript Syntax**

```javascript
Bus.stopAllEvents(
    mode
);
```
Parameters

mode
The desired stop mode. See `FMOD_STUDIO_STOP_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

- Studio::EventInstance::stop
Firelight Technologies FMOD Studio API
Studio::Bus::unlockChannelGroup

Releases the Low Level ChannelGroup locked by Studio::Bus::lockChannelGroup.

C++ Syntax

FMOD_RESULT Studio::Bus::unlockChannelGroup();

C Syntax

FMOD_RESULT FMOD_Studio_Bus_UnlockChannelGroup(FMOD_STUDIO_BUS *bus);

C# Syntax

RESULT Studio.Bus.unlockChannelGroup();

JavaScript Syntax

Bus.unlockChannelGroup();
Return Values

If the function succeeds then the return value 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 can force a bus's ChannelGroup to be created by calling `Studio::Bus::lockChannelGroup`. This function reverts to the default behavior of creating and destroying the ChannelGroup on demand, meaning it will only exist if at least one event instance routes into the bus.

Calling this function will cause the ChannelGroup to be destroyed if there are currently no event instances routing into the bus.
See Also

- Studio::Bus::lockChannelGroup
- Studio::Bus::getChannelGroup
Firelight Technologies FMOD Studio API
**Studio::VCA**

Represents a global mixer VCA.
Functions

Studio::VCA::getID  Studio::VCA::getPath
Studio::VCA::getVolume
Studio::VCA::setVolume
See Also

- `Studio::System::getVCA`
Firelight Technologies FMOD Studio API
**Studio::VCA::getID**

Retrieves the ID of the VCA.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::VCA::getID(
    FMOD_GUID *id
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_VCA_GetID(
    FMOD_STUDIO_VCA *vca,
    FMOD_GUID *id
);
```

**C# Syntax**

```csharp
RESULT Studio.VCA.getID(
    out Guid id
);
```

**JavaScript Syntax**

```javascript
VCA.getID(
    id, // writes value to id.val
);
```
Parameters

id

Address of a variable to receive the 128-bit GUID.
Return Values

If the function succeeds then the return 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 Studio API
Studio::VCA::getPath

Retrieves the path of the VCA.

C++ Syntax

`FMOD_RESULT Studio::VCA::getPath(
    char *path,
    int size,
    int *retrieved
);`

C Syntax

`FMOD_RESULT FMOD_Studio_VCA_GetPath(
    FMOD_STUDIO_VCA *vca,
    char *path,
    int size,
    int *retrieved
);`

C# Syntax

`RESULT Studio.VCA.getPath(
    out string path
);`

JavaScript Syntax

`VCA.getPath(
    path, // writes value to path.val
    size,
    retrieved // writes value to retrieved.val
);`
Parameters

path
Address of a buffer to receive the path. Specify 0 or NULL to ignore.

size
Size of the path buffer in bytes. Required if path parameter is not NULL.

retrieved
Address of a variable to receive the size of the retrieved path in bytes, including the terminating null character. 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 will return **FMOD_ERR_EVENT_NOTFOUND** unless string data for the requested VCA is loaded (by loading the "Master Bank.strings.bank" file).

If the retrieved path is too long to fit in the supplied buffer, it will be truncated and this function will return **FMOD_ERR_TRUNCATED**. Use the `retrieved` parameter to obtain the minimum buffer size required to hold the full path.

**JavaScript only :**

Note: For the "path" parameter, the maximum string length is 512.
Firelight Technologies FMOD Studio API
Studio::VCA::getVolume

Retrieves the volume level of the VCA.

C++ Syntax

FMOD_RESULT Studio::VCA::getVolume(
    float *volume,
    float *finalvolume
);

C Syntax

FMOD_RESULT FMOD_Studio_VCA_GetVolume(
    FMOD_STUDIO_VCA *vca,
    float *volume,
    float *finalvolume
);

C# Syntax

RESULT Studio.VCA.getVolume(
    out float volume,
    out float finalvolume
);

JavaScript Syntax

VCA.getVolume(
    volume, // writes value to volume.val
    finalvolume // writes value to finalvolume.val
);
Parameters

volume
   Address of a variable to receive the volume as set from the public API. Specify 0 or NULL to ignore.
finalvolume
   Address of a variable to receive the final combined volume. 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 can return both the volume set from the last call to
Studio::VCA::setVolume, as well as the final combined volume which may be
the result of automation and modulation.

NOTE: The final volume will not change immediately. It will change during
subsequent asynchronous updates.
See Also

- Studio::VCA::setVolume
Firelight Technologies FMOD Studio API
Studio::VCA::setVolume

Sets the volume level of the VCA.

C++ Syntax

```cpp
FMOD_RESULT Studio::VCA::setVolume(
    float volume
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_VCA_SetVolume(
    FMOD_STUDIO_VCA *vca,
    float volume
);
```

C# Syntax

```csharp
RESULT Studio.VCA.setVolume(
    float volume
);
```

JavaScript Syntax

```javascript
VCA.setVolume(
    volume
);
```
Parameters

volume
The volume level to set as a linear gain. 0 = silent, 1 = 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 allows volume control of all audio controlled by the VCA. This fader level is relative to the fader level set in FMOD Studio.
See Also

- Studio::VCA::getVolume
Firelight Technologies FMOD Studio API
Studio::Bank

A bank contains FMOD Studio Event data as well as the actual sound data for all events in that bank.
Functions

Studio::Bank::getBusCount  Studio::Bank::getBusList
Studio::Bank::getEventCount
Studio::Bank::getEventList
Studio::Bank::getID
Studio::Bank::getLoadingState
Studio::Bank::getPath
Studio::Bank::getSampleLoadingState
Studio::Bank::getStringCount
Studio::Bank::getStringInfo
Studio::Bank::getUserData
Studio::Bank::getVCACount
Studio::Bank::getVCAList
Studio::Bank::loadSampleData
Studio::Bank::setUserData
Studio::Bank::unload
Studio::Bank::unloadSampleData
See Also

- Studio::System::loadBankFile
- Studio::System::loadBankMemory
- Studio::System::loadBankCustom
Firelight Technologies FMOD Studio API
**Studio::Bank::getBusCount**

Retrieves the number of buses in the bank.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bank::getBusCount(
    int *count
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bank_GetBusCount(
    FMOD_STUDIO_BANK *bank,
    int *count
);
```

**C# Syntax**

```csharp
RESULT Studio.Bank.getBusCount(
    out int count
);
```

**JavaScript Syntax**

```javascript
Bank.getBusCount(
    count
    // writes value to count.val
);```
Parameters

count
   Address of a variable to receive the number of buses.
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

Used in conjunction with Studio::Bank::getBusList to enumerate the buses.
See Also

- Studio::Bank::getBusList
Firelight Technologies FMOD Studio API
Studio::Bank::getBusList

Retrieves the buses in the bank.

C++ Syntax

```cpp
FMOD_RESULT Studio::Bank::getBusList(
    Studio::Bus **array,
    int capacity,
    int *count
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_Bank_GetBusList(
    FMOD_STUDIO_BANK *bank,
    FMOD_STUDIO_BUS **array,
    int capacity,
    int *count
);
```

C# Syntax

```csharp
RESULT Studio.Bank.getBusList(
    out Bus[] array
);
```

JavaScript Syntax

```javascript
Bank.getBusList(
    array, // writes value to array.val
    capacity, // writes value to count.val
    count
);
```
Parameters

array
   An array of memory allocated by the user.
capacity
   The capacity of the array passed in as the first parameter
count
   Address of a variable to receive the number of buses written to 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` enum.
Remarks

Used in conjunction with `Studio::Bank::getBusCount` to enumerate the buses.
See Also

- Studio::Bank::getBusCount
Firelight Technologies FMOD Studio API
**Studio::Bank::getEventCount**

Retrieves the number of EventDescriptions in the bank.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bank::getEventCount(
    int *count
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bank_GetEventCount(
    FMOD_STUDIO_BANK *bank,
    int *count
);
```

**C# Syntax**

```csharp
RESULT Studio.Bank.getEventCount(
    out int count
);
```

**JavaScript Syntax**

```javascript
Bank.getEventCount(
    count // writes value to count.val
);
```
Parameters

count
    Address of a variable to receive the number of EventDescriptions.
Return Values

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

Can be used in conjunction with Studio::Bank::getEventList to enumerate the events.

**NOTE!** This function only counts events that have been added explicitly to the bank by the sound designer. Events that have been implicitly included via event instrument references are not counted.
See Also

- Studio::Bank::getEventList
Firelight Technologies FMOD Studio API
Studio::Bank::getEventList

Retrieves the EventDescriptions in the bank.

C++ Syntax

```c++
FMOD_RESULT Studio::Bank::getEventList(
    Studio::EventDescription **array,
    int capacity,
    int *count
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_Bank_GetEventList(
    FMOD_STUDIO_BANK *bank,
    FMOD_STUDIO_EVENTDESCRIPTION **array,
    int capacity,
    int *count
);
```

C# Syntax

```csharp
RESULT Studio.Bank.getEventList(
    out EventDescription[] array
);
```

JavaScript Syntax

```javascript
Bank.getEventList(
    array, // writes value to array.val
capacity,
    count // writes value to count.val)
```
Parameters

array
   An array of memory allocated by the user.
capacity
   The capacity of the array passed in as the first parameter
count
   Address of a variable to receive the number of Event Descriptions written to 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

Can be used in conjunction with Studio::Bank::getEventCount to enumerate the events.

**NOTE!** This function only counts events that have been added explicitly to the bank by the sound designer. Events that have been implicitly included via event instrument references are not counted.
See Also

- Studio::Bank::getEventCount
Firelight Technologies FMOD Studio API
Studio::Bank::getID

Retrieves the ID of the bank.

C++ Syntax

FMOD_RESULT Studio::Bank::getID(
    FMOD_GUID *id
);

C Syntax

FMOD_RESULT FMOD_Studio_Bank_GetID(
    FMOD_STUDIO_BANK *bank,
    FMOD_GUID *id
);

C# Syntax

RESULT Studio.Bank.getID(
    out Guid id
);

JavaScript Syntax

Bank.getID(
    id
    // writes value to id.val
);
Parameters

id
   Address of a variable to receive the ID.
Return Values

If the function succeeds then the return 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 Studio API
Studio::Bank::getLoadingState

Retrieves the bank loading state.

C++ Syntax

FMOD_RESULT Studio::Bank::getLoadingState(
    FMOD_STUDIO_LOADING_STATE *state
);

C Syntax

FMOD_RESULT FMOD_Studio_Bank_GetLoadingState(
    FMOD_STUDIO_BANK *bank,
    FMOD_STUDIO_LOADING_STATE *state
);

C# Syntax

RESULT Studio.Bank.getLoadingState(
    out LOADING_STATE state
);

JavaScript Syntax

Bank.getLoadingState(
    state // writes value to state.val
);
Parameters

state
    Address of a variable to receive the loading 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

This function provides a way of tracking banks which can be loaded asynchronously. Only after a bank has finished loading is it valid to query event descriptions within the bank.

If the bank has a file error, then the state will be returned as `FMOD_STUDIO_LOADING_STATE_ERROR` and the return code from this function will be the error code of the bank load function.

If the bank is invalid, then the state will be set to `FMOD_STUDIO_LOADING_STATE_UNLOADED`. 
See Also

- FMOD_STUDIO_LOADING_STATE
- Studio::System::loadBankFile
- Studio::System::loadBankMemory
- Studio::System::loadBankCustom
Firelight Technologies FMOD Studio API
Studio::Bank::getPath

Retrieves the path of the bank.

C++ Syntax

FMOD_RESULT Studio::Bank::getPath(
    char *path,
    int size,
    int *retrieved
);

C Syntax

FMOD_RESULT FMOD_Studio_Bank_GetPath(
    FMOD_STUDIO_BANK *bank,
    char *path,
    int size,
    int *retrieved
);

C# Syntax

RESULT Studio.Bank.getPath(
    out string path
);

JavaScript Syntax

Bank.getPath(
    path, // writes value to path.val
    size,
    retrieved // writes value to retrieved.val
);
Parameters

path
  Address of a buffer to receive the path. Specify 0 or NULL to ignore.

size
  Size of the path buffer in bytes. Required if path parameter is not NULL.

retrieved
  Address of a variable to receive the size of the retrieved path in bytes,
  including the terminating null character. Optional. Specify 0 or NULL to
  ignore.
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 function will return `FMOD_ERR_EVENT_NOTFOUND` unless string data for the requested bank is loaded (by loading the "Master Bank.strings.bank" file).

The bank path is always of the form 'bank:/name'. It is not the same as the filename used to load the bank.

If the retrieved path is too long to fit in the supplied buffer, it will be truncated and this function will return `FMOD_ERR_TRUNCATED`. Use the `retrieved` parameter to obtain the minimum buffer size required to hold the full path.

JavaScript only:

Note: For the "path" parameter, the maximum string length is 512.
Firelight Technologies FMOD Studio API
Studio::Bank::getSampleLoadingState

Retrieves the sample data loading state of the bank.

C++ Syntax

```cpp
FMOD_RESULT Studio::Bank::getSampleLoadingState(
    FMOD_STUDIO_LOADING_STATE *state
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_Bank_GetSampleLoadingState(
    FMOD_STUDIO_BANK *bank,
    FMOD_STUDIO_LOADING_STATE *state
);
```

C# Syntax

```csharp
RESULT Studio.Bank.getSampleLoadingState(
    out LOADING_STATE state
);
```

JavaScript Syntax

```javascript
Bank.getSampleLoadingState(
    state // writes value to state.val
);
```
Parameters

state
   Address of a variable to receive the loading 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

Used for tracking the status of the Studio::Bank::loadSampleData operation.

Note that this function is not affected by Studio::EventDescription::loadSampleData, so it can report FMOD_STUDIO_LOADING_STATE_UNLOADED even if all events in a bank have had their sample data loaded individually.

If the bank is invalid, then the state will be set to FMOD_STUDIO_LOADING_STATE_UNLOADED.
See Also

- `FMOD_STUDIO_LOADING_STATE`
- `Studio::Bank::loadSampleData`
Firelight Technologies FMOD Studio API
Studio::Bank::getStringCount

Retrieves the number of string table entries in the bank.

C++ Syntax

```cpp
FMOD_RESULT Studio::Bank::getStringCount(
    int *count
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_Bank_GetStringCount(
    FMOD_STUDIO_BANK *bank,
    int *count
);
```

C# Syntax

```csharp
RESULT Studio.Bank.getStringCount(
    out int count
);
```

JavaScript Syntax

```javascript
Bank.getStringCount(
    count  // writes value to count.val
);
```
Parameters

count
    Address of a variable to receive the number of string table entries.
Return Values

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

Can be used in conjunction with `Studio::Bank::getStringInfo` to directly access the string table. Only the strings bank will contain string table entries.
See Also

- Studio::Bank::getStringInfo
Firelight Technologies FMOD Studio API
**Studio::Bank::getStringInfo**

Retrieves the string table entry for the given index.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bank::getStringInfo(
    int index,
    FMOD_GUID *id,
    char *path,
    int size,
    int *retrieved
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bank_GetStringInfo(
    FMOD_STUDIO_BANK *bank,
    int index,
    FMOD_GUID *id,
    char *path,
    int size,
    int *retrieved
);
```

**C# Syntax**

```csharp
RESULT Studio.Bank.getStringInfo(
    int index,
    out Guid id,
    out string path
);
```

**JavaScript Syntax**

```javascript
Bank.getStringInfo(
    index, // writes value to id.val
    id,
    path, // writes value to path.val
    size,
    retrieved // writes value to retrieved.val
);
```
Parameters

index
   Index of string table entry to retrieve.

id
   Address of a variable to receive the ID. Specify 0 or NULL to ignore.

path
   Address of a buffer to receive the path. Specify 0 or NULL to ignore.

size
   Size of the path buffer in bytes. Required if path parameter is not NULL.

retrieved
   Address of a variable to receive the size of the retrieved path in bytes, including the terminating null character. 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 string table contains mappings from ID to path and is automatically used by Studio::System::lookupID and Studio::System::lookupPath. This function can be called to get the string data directly.

If the retrieved path is too long to fit in the supplied buffer, it will be truncated and this function will return FMOD_ERR_TRUNCATED. Use the retrieved parameter to obtain the minimum buffer size required to hold the full path.

JavaScript only:

Note: For the "path" parameter, the maximum string length is 512.
See Also

- Studio::Bank::getStringCount
Firelight Technologies FMOD Studio API
**Studio::Bank::getUserData**

Retrieves the user data that is set on the bank.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bank::getUserData(
    void **userData
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bank_GetUserData(
    FMOD_STUDIO_BANK *bank,
    void **userData
);
```

**C# Syntax**

```csharp
RESULT Studio.Bank.getUserData(
    out IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
Bank.getUserData(
    userdata                                            // writes value to userdata.val
);
```
Parameters

userData
    Address of a variable to receive the 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.
See Also

- Studio::Bank::setUserData
Firelight Technologies FMOD Studio API
Studio::Bank::getVCACount

Retrieves the number of VCAs in the bank.

C++ Syntax

FMOD_RESULT Studio::Bank::getVCACount(
    int *count
);

C Syntax

FMOD_RESULT FMOD_Studio_Bank_GetVCACount(
    FMOD_STUDIO_BANK *bank,
    int *count
);

C# Syntax

RESULT Studio.Bank.getVCACount(
    out int count
);

JavaScript Syntax

Bank.getVCACount(
    count // writes value to count.val
);
Parameters

count
   Address of a variable to receive the number of VCAs.
Return Values

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

Used in conjunction with Studio::Bank::getVCAList to enumerate the VCAs.
See Also

- Studio::Bank::getVCAList
Firelight Technologies FMOD Studio API
**Studio::Bank::getVCAList**

Retrieves the VCAs in the bank.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bank::getVCAList(
    Studio::VCA **array,
    int capacity,
    int *count
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bank_GetVCAList(
    FMOD_STUDIO_BANK *bank,
    FMOD_STUDIO_VCA **array,
    int capacity,
    int *count
);
```

**C# Syntax**

```csharp
RESULT Studio.Bank.getVCAList(
    out VCA[] array
);
```

**JavaScript Syntax**

```javascript
Bank.getVCAList(
    array,                      // writes value to array.val
capacity,
    count                       // writes value to count.val
);
```
Parameters

array
   An array of memory allocated by the user.
capacity
   The capacity of the array passed in as the first parameter
count
   Address of a variable to receive the number of VCAs written to the array
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

Used in conjunction with Studio::Bank::getVCACount to enumerate the VCAs.
See Also

- Studio::Bank::getVCACount
Firelight Technologies FMOD Studio API
**Studio::Bank::loadSampleData**

Loads all non-streaming sample data used by events in the bank.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bank::loadSampleData();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bank_LoadSampleData(FMOD_STUDIO_BANK *bank);
```

**C# Syntax**

```csharp
RESULT Studio.Bank.loadSampleData();
```

**JavaScript Syntax**

```javascript
Bank.loadSampleData();
```
Return Values

If the function succeeds then the return value 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 can use this function to preload sample data ahead of time so that events can be played immediately when required.

Each time this function is called, it will increment the reference count, so the sample data will not be unloaded until `Studio::Bank::unloadSampleData` is called the same number of times.

It is valid to mix calls to `Studio::Bank::loadSampleData` with calls to `Studio::EventDescription::loadSampleData`. If you do this, the sample data will be loaded when either reference count is non-zero, and will be unloaded when both reference counts go to zero.
See Also

- Studio::Bank::unloadSampleData
- Studio::Bank::getSampleLoadingState
- Studio::EventDescription::loadSampleData
- Studio::EventDescription::unloadSampleData
**Studio::Bank::setUserData**

Sets arbitrary user data on the bank.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bank::setUserData(
    void *userData
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bank_SetUserData(
    FMOD_STUDIO_BANK *bank,
    void *userData
);
```

**C# Syntax**

```csharp
RESULT Studio.Bank.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
Bank.setUserData(
    userdata
);
```
Parameters

userData
  Address of user data to store within the event description 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 (e.g. if many objects of this type exist). Using Studio::Bank::getUserData would help in the identification of the object.
See Also

- Studio::Bank::getUserData
Firelight Technologies FMOD Studio API
Studio::Bank::unload

Unloads the bank and all of its data.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::Bank::unload();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_Bank_Unload(FMOD_STUDIO_BANK *bank);
```

**C# Syntax**

```csharp
RESULT Studio.Bank.unload();
```

**JavaScript Syntax**

```javascript
Bank.unload();
```
Return Values

If the function succeeds then the return value 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 destroy all objects created from the bank, unload all sample data inside the bank, and invalidate all API handles referring to the bank.
See Also

- Studio::System::loadBankFile
- Studio::System::loadBankMemory
- Studio::System::loadBankCustom
Firelight Technologies FMOD Studio API
Studio::Bank::unloadSampleData

Unloads all non-streaming sample data used by events in the bank.

C++ Syntax

```cpp
FMOD_RESULT Studio::Bank::unloadSampleData();
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_Bank_UnloadSampleData(FMOD_STUDIO_BANK *bank);
```

C# Syntax

```csharp
RESULT Studio.Bank.unloadSampleData();
```

JavaScript Syntax

```javascript
Bank.unloadSampleData();
```
Return Values

If the function succeeds then the return value 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 time this function is called, it will decrement the reference count. If the reference count goes to zero, the sample data will be unloaded.

Any sample data that is being used by event instances will not be unloaded until the event instances are released.
See Also

- Studio::Bank::loadSampleData
- Studio::Bank::getSampleLoadingState
- Studio::EventDescription::loadSampleData
- Studio::EventDescription::unloadSampleData
Firelight Technologies FMOD Studio API
Studio::CommandReplay
Functions

Studio::CommandReplay::getCommandAtTime
Studio::CommandReplay::getCommandCount
Studio::CommandReplay::getCommandInfo
Studio::CommandReplay::getCommandString
Studio::CommandReplay::getCurrentCommand
Studio::CommandReplay::getLength
Studio::CommandReplay::getPaused
Studio::CommandReplay::getPlaybackState
Studio::CommandReplay::getSystem
Studio::CommandReplay::getUserData
Studio::CommandReplay::release
Studio::CommandReplay::seekToCommand
Studio::CommandReplay::seekToTime
Studio::CommandReplay::setBankPath
Studio::CommandReplay::setCreateInstanceCallback
Studio::CommandReplay::setFrameCallback
Studio::CommandReplay::setLoadBankCallback
Studio::CommandReplay::setPaused
Studio::CommandReplay::setUserData
Studio::CommandReplay::start
Studio::CommandReplay::stop
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getCommandAtTime

Finds a command that corresponds to the given playback time.

C++ Syntax

FMOD_RESULT Studio::CommandReplay::getCommandAtTime(
    float time,
    int *commandIndex
);

C Syntax

FMOD_RESULT FMOD_Studio_CommandReplay_GetCommandAtTime(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    float time,
    int *commandIndex
);

C# Syntax

RESULT Studio.CommandReplay.getCommandAtTime(
    float time,
    out int commandIndex
);

JavaScript Syntax

CommandReplay.getCommandAtTime(
    time,
    commandindex // writes value to commandindex.v
);
Parameters

time
  The time used to find a command index.
commandIndex
  Address of the variable to receive the command 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

The time passed in must be less than or equal to the total playback time of the replay. It will return an index for the first command which has an equal or greater time.
See Also

- Studio::CommandReplay::getCommandCount
- Studio::CommandReplay::getCommandString
- Studio::CommandReplay::getCommandInfo
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getCommandCount

Retrieves the number of commands in the replay.

C++ Syntax

```cpp
FMOD_RESULT Studio::CommandReplay::getCommandCount(
    int *count
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_CommandReplay_GetCommandCount(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    int *count
);
```

C# Syntax

```csharp
RESULT Studio.CommandReplay.getCommandCount(
    out int count
);
```

JavaScript Syntax

```javascript
CommandReplay.getCommandCount(
    count
); // writes value to count.val
```
Parameters

count
  Address of the variable to receive the count.
Return Values

If the function succeeds then the return 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

- Studio::CommandReplay::getCommandInfo
- Studio::CommandReplay::getCommandString
- Studio::CommandReplay::getCommandAtTime
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getCommandInfo

Retrieves information about the command at the given index.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::getCommandInfo(
    int commandIndex,
    FMOD_STUDIO_COMMAND_INFO *info
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_GetCommandInfo(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    int commandIndex,
    FMOD_STUDIO_COMMAND_INFO *info
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.getCommandInfo(
    int commandIndex,
    out COMMAND_INFO info
);
```

**JavaScript Syntax**

```javascript
CommandReplay.getCommandInfo(
    commandindex,
    info // writes value to info.val
);
```
Parameters

commandIndex
   The index of the command.
info
   Address of the variable to receive command info structure.
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

- Studio::CommandReplay::getCommandCount
- Studio::CommandReplay::getCommandString
- Studio::CommandReplay::getCommandAtTime
- FMOD_STUDIO_COMMAND_INFO
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getCommandString

Retrieves a text string representation of the command at the given index.

C++ Syntax

```cpp
FMOD_RESULT Studio::CommandReplay::getCommandString(
    int commandIndex,
    char *buffer,
    int length
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_CommandReplay_GetCommandString(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    int commandIndex,
    char *buffer,
    int length
);
```

C# Syntax

```csharp
RESULT Studio.CommandReplay.getCommandString(
    int commandIndex,
    out string description
);
```

JavaScript Syntax

```javascript
CommandReplay.getCommandString(
    commandindex,
    buffer // writes value to buffer.val
);
```
Parameters

commandIndex
  The index of the command.
buffer
  Address of the variable to receive the string.
length
  The capacity 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

If the retrieved path is too long to fit in the supplied buffer, it will be truncated and this function will return `FMOD_ERR_TRUNCATED`.

JavaScript only:

Note: For the "buffer" parameter, the maximum string length is 512.
See Also

- Studio::CommandReplay::getCommandCount
- Studio::CommandReplay::getCommandInfo
- Studio::CommandReplay::getCommandAtTime
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getCurrentCommand

Retrieves the progress through the command replay.

C++ Syntax

FMOD_RESULT Studio::CommandReplay::getCurrentCommand(
  int *commandIndex,
  float *currentTime
);

C Syntax

FMOD_RESULT FMOD_Studio_CommandReplay_GetCurrentCommand(
  FMOD_STUDIO_COMMANDREPLAY *commandreplay,
  int *commandIndex,
  float *currentTime
);

C# Syntax

RESULT Studio.CommandReplay.getCurrentCommand(
  out int commandIndex,
  out float currentTime
);

JavaScript Syntax

CommandReplay.getCurrentCommand(
  commandindex, // writes value to commandindex.v
  currenttime, // writes value to currenttime.va
);
Parameters

commandIndex
   The address of the variable to hold the current command index. Specify 0 or NULL to ignore.

currentTime
   The address of the variable to hold the current playback time. 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 returns the current command index the playback is up to, and the current playback time.
See Also

- Studio::CommandReplay::start
- Studio::CommandReplay::stop
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getLength

Retrieves the total playback time of the command replay.

C++ Syntax

FMOD_RESULT Studio::CommandReplay::getLength(
    float *totalTime
);

C Syntax

FMOD_RESULT FMOD_Studio_CommandReplay_GetLength(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    float *totalTime
);

C# Syntax

RESULT Studio.CommandReplay.getLength(
    out float totalTime
);

JavaScript Syntax

CommandReplay.getLength(
    length // writes value to length.val
);
Parameters

totalTime
    Address of the variable to receive the total 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.
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getPaused

Retrieves the paused state of the replay.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::getPaused(
    bool *paused
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_GetPaused(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    FMOD_BOOL *paused
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.getPaused(
    out bool paused
);
```

**JavaScript Syntax**

```javascript
CommandReplay.getPaused(
    paused // writes value to paused.val
);
```
Parameters

paused

Address of the variable to hold the paused 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

- Studio::CommandReplay::setPaused
Firelight Technologies FMOD Studio API
**Studio::CommandReplay::getPlaybackState**

Retrieves the playback state.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::getPlaybackState(
    FMOD_STUDIO_PLAYBACK_STATE *state
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_GetPlaybackState(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    FMOD_STUDIO_PLAYBACK_STATE *state
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.getPlaybackState(
    out PLAYBACK_STATE state
);
```

**JavaScript Syntax**

```javascript
CommandReplay.getPlaybackState(  
    state // writes value to state.val
);
```
Parameters

state
The address of the variable to hold the playback state. 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

- Studio::CommandReplay::start
- Studio::CommandReplay::stop
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getSystem

Retrieves the system associated with this replay object.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::getSystem(
    Studio::System **system
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_GetSystem(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    FMOD_STUDIO_SYSTEM **system
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.getSystem(
    out System system
);
```

**JavaScript Syntax**

```javascript
CommandReplay.getSystem(
    system // writes value to system.val
);
```
Parameters

system
   Address of the variable to receive the 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.
Firelight Technologies FMOD Studio API
Studio::CommandReplay::getUserData

Retrieves the user data that is set on the command replay.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::getUserData(
    void **userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_GetUserData(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    void **userdata
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.getUserData(
    out IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
CommandReplay.getUserData(
    userdata // writes value to userdata.val
);
```
Parameters

userdata
Return Values

If the function succeeds then the return 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

- [Studio::CommandReplay::setUserData](#)

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
**Studio::CommandReplay::release**

Releases the command replay.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::release();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_Release(FMOD_STUDIO_COMMANDREPLAY *
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.release();
```

**JavaScript Syntax**

```javascript
CommandReplay.release();
```
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 will free all resources associated with the command replay.
Firelight Technologies FMOD Studio API
Studio::CommandReplay::seekToCommand

Seek to a given command index in the command replay.

C++ Syntax

```cpp
FMOD_RESULT Studio::CommandReplay::seekToCommand(
    int commandIndex
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_CommandReplay_SeekToCommand(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    int commandIndex
);
```

C# Syntax

```csharp
RESULT Studio.CommandReplay.seekToCommand(
    int commandIndex
);
```

JavaScript Syntax

```javascript
CommandReplay.seekToCommand(
    commandindex
);
```
Parameters

commandIndex
   The command index to seek 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

This function sets the seek command to seek to. The replay will start from the given seek position.
See Also

- Studio::CommandReplay::getCommandCount
- Studio::CommandReplay::getCommandAtTime
- Studio::CommandReplay::seekToTime
Firelight Technologies FMOD Studio API
Studio::CommandReplay::seekToTime

Seek to a given time in the command replay.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::seekToTime(
    float time
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_SeekToTime(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    float time
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.seekToTime(
    float time
);
```

**JavaScript Syntax**

```javascript
CommandReplay.seekToTime(
    time
);
```
Parameters

time
   The time to seek 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

This function sets the seek command to seek to by looking up the given time.
See Also

- Studio::CommandReplay::seekToCommand
Firelight Technologies FMOD Studio API
Studio::CommandReplay::setBankPath

Sets a path substitution that will be used when loading banks with this replay.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::setBankPath(
    const char *bankPath
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_SetBankPath(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    const char *bankPath
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.setBankPath(
    string bankPath
);
```

**JavaScript Syntax**

```javascript
CommandReplay.setBankPath(
    bankPath
);
```
Parameters

bankPath
   The path to use when loading banks.
Return Values

If the function succeeds then the return value 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 allows a replay to be redirected to load banks for a given directory, instead of the original one recorded as part of the replay file. It will only affect the `Studio::System::loadBankFile` command in the replay.
See Also

- `Studio::CommandReplay::setLoadBankCallback`
Firelight Technologies FMOD Studio API
**Studio::CommandReplay::setCreateInstanceCallback**

Sets a callback that is invoked when the replay reaches a create event instance command.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::setCreateInstanceCallback(
    FMOD_STUDIO_COMMANDREPLAY_CREATE_INSTANCE_CALLBACK callback);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_SetCreateInstanceCallback(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    FMOD_STUDIO_COMMANDREPLAY_CREATE_INSTANCE_CALLBACK callback);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.setCreateInstanceCallback(
    COMMANDREPLAY_CREATE_INSTANCE_CALLBACK callback);
```

**JavaScript Syntax**

```javascript
CommandReplay.setCreateInstanceCallback(
    callback);
```
Parameters

callback
   The callback to use.
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

The create instance bank callback is invoked whenever a `Studio::EventDescription::createInstance` command is reached.

If this callback is not set the event instance is created automatically. If the callback is set, it is up to the callback to either create a new instance or return without creating anything. If the instance is not created, then subsequent event instance calls will be ignored in the replay.

The callback contains the original handle that was created when the replay was recorded. This will not correspond to any valid object, but it can be used as a way of identifying different instances in the replay.
See Also

- FMOD_STUDIO_COMMANDREPLAY_CREATE_INSTANCE_CALLBACK
Firelight Technologies FMOD Studio API
Studio::CommandReplay::setFrameCallback

Sets a callback that is issued when the replay reaches a new frame.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::setFrameCallback(
    FMOD_STUDIO_COMMANDREPLAY_FRAME_CALLBACK callback);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_SetFrameCallback(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    FMOD_STUDIO_COMMANDREPLAY_FRAME_CALLBACK callback);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay setFrameCallback(
    COMMANDREPLAY_FRAME_CALLBACK callback);
```

**JavaScript Syntax**

```javascript
CommandReplay.setFrameCallback(callback);
```
Parameters

callback
   The callback to use.
Return Values

If the function succeeds then the return value 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 frame callback is called whenever a new frame of the replay is reached. Each frame command will advance the time of the replay.
See Also

- FMOD_STUDIO_COMMANDREPLAY_FRAME_CALLBACK
Firelight Technologies FMOD Studio API
Studio::CommandReplay::setLoadBankCallback

Sets a callback that is invoked when the replay reaches a load bank command.

C++ Syntax

FMOD_RESULT Studio::CommandReplay::setLoadBankCallback(
    FMOD_STUDIO_COMMANDREPLAY_LOAD_BANK_CALLBACK callback
);

C Syntax

FMOD_RESULT FMOD_Studio_CommandReplay_SetLoadBankCallback(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    FMOD_STUDIO_COMMANDREPLAY_LOAD_BANK_CALLBACK callback
);

C# Syntax

RESULT Studio.CommandReplay.setLoadBankCallback(
    COMMANDREPLAY_LOAD_BANK_CALLBACK callback
);

JavaScript Syntax

CommandReplay.setLoadBankCallback(
    callback
);
Parameters

callback
   The callback to use.
Return Values

If the function succeeds then the return value 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 load bank callback is invoked whenever any of the Studio load bank functions are reached. For \texttt{Studio::System::loadBankFile}, the callback will have the filename used in the replay file, modified by any path set in \texttt{Studio::CommandReplay::setBankPath}. For banks loaded with \texttt{Studio::System::loadBankMemory} and \texttt{Studio::System::loadBankCustom}, the path will be NULL but the guid can be used to identify what bank was loaded.

If this callback is not set the bank is loaded automatically. If the callback is set the bank is not loaded automatically and should be loaded in the callback. Failing to load the bank in the callback will mean the replay will continue but some later events may not be found.
See Also

- FMOD_STUDIO_COMMANDREPLAY_LOAD_BANK_CALLBACK
Firelight Technologies FMOD Studio API
**Studio::CommandReplay::setPaused**

Sets the replay paused or unpaged.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::setPaused(
    bool paused
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_SetPaused(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    FMOD_BOOL paused
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.setPaused(
    bool paused
);
```

**JavaScript Syntax**

```javascript
CommandReplay.setPaused(
    paused
);
```
Parameters

paused
   The desired pause state. true = pause, 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.
Remarks

When paused the replay will no longer continue to issue replay commands.
See Also

- Studio::CommandReplay::setPaused
Firelight Technologies FMOD Studio API
Studio::CommandReplay::setUserData

Sets arbitrary user data on the command replay.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::setUserData(
    void *userdata
);
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_SetUserData(
    FMOD_STUDIO_COMMANDREPLAY *commandreplay,
    void *userdata
);
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.setUserData(
    IntPtr userdata
);
```

**JavaScript Syntax**

```javascript
CommandReplay.setUserData(
    userdata
);
```
Parameters

userdata
Return Values

If the function succeeds then the return value 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

- Studio::CommandReplay::setUserData
Firelight Technologies FMOD Studio API
**Studio::CommandReplay::start**

Starts the command replay running.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::start();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_Start(FMOD_STUDIO_COMMANDREPLA
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.start();
```

**JavaScript Syntax**

```javascript
CommandReplay.start();
```
Return Values

If the function succeeds then the return value 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 starts the replay running. The state of the replay can be called from Studio::CommandReplay::getPlaybackState.

If the replay is already running then it will restart from the beginning.

Once the replay finishes it will automatically stop.
See Also

- Studio::System::loadCommandReplay
- Studio::CommandReplay::stop
- Studio::CommandReplay::getPlaybackState

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
**Studio::CommandReplay::stop**

Stops the command replay.

**C++ Syntax**

```cpp
FMOD_RESULT Studio::CommandReplay::stop();
```

**C Syntax**

```c
FMOD_RESULT FMOD_Studio_CommandReplay_Stop(FMOD_STUDIO_COMMANDREPLAY
```

**C# Syntax**

```csharp
RESULT Studio.CommandReplay.stop();
```

**JavaScript Syntax**

```javascript
CommandReplay.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 stops the replay.

Unless the `FMOD_STUDIO_COMMANDREPLAY_SKIP_CLEANUP` flag has been used, all resources that were created as part of the replay will be removed when the replay stops.
See Also

- Studio::CommandReplay::start
- Studio::CommandReplay::getPlaybackState
Firelight Technologies FMOD Studio API
Functions

ParseID
Firelight Technologies FMOD Studio API
Studio::parseID

Parses an ID (a 128-bit GUID) from a string.

C++ Syntax

```cpp
FMOD_RESULT Studio::parseID(
    const char *idString,
    FMOD_GUID *id
);
```

C Syntax

```c
FMOD_RESULT FMOD_Studio_ParseID(
    const char *idString,
    FMOD_GUID *id
);
```

C# Syntax

```csharp
static RESULT Studio.Util.ParseID(
    string idString,
    out Guid id
);
```
Parameters

idString

The string representation of the ID.

id

Address of a variable to receive the parsed ID.
Return Values

If the function succeeds then the return value 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 expects a string representation of a GUID in the following exact format: "\{9d348364-8145-4724-b337-5bc9b2afe60f\}".
See Also

- Studio::System::getEvent
- Studio::System::getBus
- Studio::System::getVCA
- Studio::System::getBank
Firelight Technologies FMOD Studio API
Callbacks

FMOD_STUDIO_COMMANDREPLAY_CREATE_INSTANCE_CALLBACK
FMOD_STUDIO_COMMANDREPLAY_FRAME_CALLBACK
FMOD_STUDIO_COMMANDREPLAY_LOAD_BANK_CALLBACK
FMOD_STUDIO_EVENT_CALLBACK
FMOD_STUDIO_SYSTEM_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_STUDIO_COMMANDREPLAY_CREATE_INSTANCE_CALLBACK

Callback for command replay event instance creation.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_STUDIO_COMMANDREPLAY_CREATE_INSTANCE_CALLBACK(  
    FMOD_STUDIO_COMMANDREPLAY *replay,  
    int commandIndex,  
    FMOD_STUDIO_EVENTDESCRIPTION *eventDescription,  
    FMOD_STUDIO_EVENTINSTANCE **instance,  
    void *userdata
);
```
**Parameters**

*replay*

Pointer to the command replay object.

*commandIndex*

The command that invoked this callback.

*eventDescription*

The event description to use.

*instance*

The resulting event instance.

*userdata*

The userdata assigned into the given replay, or NULL if not 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

C++ Users. Cast `FMOD_STUDIO_COMMANDREPLAY *` to `FMOD::Studio::CommandReplay *` inside the callback and use as normal.

'userdata' is the userdata assigned to the system from `Studio::CommandReplay::setUserData` function.

This callback can be used to control event instance creation in a command replay.

**NOTE!** The original handle does not represent a valid object. It should only used to match against the replay file as a unique identifier.
See Also

- Studio::CommandReplay::setCreateInstanceCallback
Firelight Technologies FMOD Studio API
FMOD_STUDIO_COMMANDREPLAY_FRAME_CALLBACK

Callback for when the command replay goes to the next frame.

C/C++ Syntax

```c
FMOD_RESULT F_CALLBACK FMOD_STUDIO_COMMANDREPLAY_FRAME_CALLBACK(
    FMOD_STUDIO_COMMANDREPLAY *replay,
    int commandIndex,
    float currentTime,
    void *userdata
);
```
Parameters

replay
Pointer to the command replay object.

commandIndex
The current playback command index.

currentTime
The current playback time.

userdata
The userdata assigned into the given replay, or NULL if not 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

C++ Users. Cast `FMOD_STUDIO_COMMANDREPLAY *` to `FMOD::Studio::CommandReplay *` inside the callback and use as normal.

'userdata' is the userdata assigned to the system from `Studio::CommandReplay::setUserData` function.

The command replay file is batched into frames, where each frame will be executed only after the appropriate time as passed. Each time a frame is passed this callback will be invoked.
See Also

- Studio::CommandReplay::setFrameCallback
Firelight Technologies FMOD Studio API
FMOD_STUDIO_COMMANDREPLAY_LOAD_BANK_CALLBACK

Callback for command replay bank loading.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_STUDIO_COMMANDREPLAY_LOAD_BANK_CALLBACK(
    FMOD_STUDIO_COMMANDREPLAY *replay,
    int commandIndex,
    const FMOD_GUID *bankGuid,
    const char *bankFilename,
    FMOD_STUDIO_LOAD_BANK_FLAGS flags,
    FMOD_STUDIO_BANK **bank,
    void *userdata
);


**Parameters**

`replay`

Pointer to the command replay object.

`commandIndex`

The command that invoked this callback.

`bankGuid`

The guid of the bank that needs to be loaded. May be all zero if not known.

`bankFilename`

The filename of the bank that needs to be loaded. May be NULL if not known.

`flags`

The flags to load the bank with.

`bank`

The resulting bank handle.

`userdata`

The userdata assigned into the given replay, or NULL if not 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

C++ Users. Cast FMOD_STUDIO_COMMANDREPLAY * to FMOD::Studio::CommandReplay * inside the callback and use as normal.

'userdata' is the userdata assigned to the system from Studio::CommandReplay::setUserData function.

This callback can be used to load banks when playing a command replay.
See Also

- [Studio::CommandReplay::setLoadBankCallback](#)
Firelight Technologies FMOD Studio API
**FMOD_STUDIO_EVENT_CALLBACK**

Callback that is fired when a Studio::EventInstance changes state.

**C/C++ Syntax**

```c
FMOD_RESULT F_CALLBACK FMOD_STUDIO_EVENT_CALLBACK(
    FMOD_STUDIO_EVENT_CALLBACK_TYPE type,
    FMOD_STUDIO_EVENTINSTANCE *event,
    void *parameters
);
```
Parameters

*type*

The type of event that has occurred.

*event*

The event instance that has changed state. Can be cast to Studio::EventInstance*

type.

*parameters*

The callback parameters. The data passed varies based on the callback 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 callback is used for tracking replay state and providing programmer sounds. The data passed to the callback function in the parameters argument varies based on the callback type. See FMOD_STUDIO_EVENT_CALLBACK_TYPE for more information.

Example:

```cpp
def F_CALLBACK MyCallback(FMOD_STUDIO_EVENT_CALLBACK_TYPE type, FMOD_STUDIO_EVENTINSTANCE *event):
    instance = (Studio::EventInstance *)(event)
    if (type == FMOD_STUDIO_EVENT_CALLBACK_STOPPED):
        # Handle event instance stop here
    else if (type == FMOD_STUDIO_EVENT_CALLBACK_CREATE_PROGRAMMER_SOUND):
        properties = (FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES)
        # Handle programmer sound creation here
    else if (type == FMOD_STUDIO_EVENT_CALLBACK_DESTROY_PROGRAMMER_SOUND):
        properties = (FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES)
        # Handle programmer sound destruction here
    return FMOD_OK
```

See Also

- Studio::EventInstance::setCallback
- Studio::EventDescription::setCallback
- Studio::System::getSoundInfo
- FMOD_STUDIO_EVENT_CALLBACK_TYPE
- FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES
Firelight Technologies FMOD Studio API
FMOD_STUDIO_SYSTEM_CALLBACK

Callback for Studio system events.

C/C++ Syntax

FMOD_RESULT F_CALLBACK FMOD_STUDIO_SYSTEM_CALLBACK(
    FMOD_STUDIO_SYSTEM *system,
    FMOD_STUDIO_SYSTEM_CALLBACK_TYPE type,
    void *commanddata,
    void *userdata
);
Parameters

system
Pointer to a Studio system.

type
The type of callback. Refer to \texttt{FMOD\_STUDIO\_SYSTEM\_CALLBACK\_TYPE}.

commanddata
The callback type specific data generated by the callback. See remarks for meaning.

userdata
The userdata assigned into the given system, or NULL if not 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

C++ Users. Cast FMOD_STUDIO_SYSTEM * to FMOD::Studio::System * inside the callback and use as normal.

'userdata' is the userdata assigned to the system from Studio::System::setUserData function.

'commanddata' is not currently used for any callback type. It is included for future compatibility.
See Also

- Studio::System::setCallback
- FMOD_STUDIO_SYSTEM_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
Structures

FMOD_STUDIO_ADVANCEDSETTINGS
FMOD_STUDIO_BANK_INFO
FMOD_STUDIO_BUFFER_INFO
FMOD_STUDIO_BUFFER_USAGE
FMOD_STUDIO_COMMAND_INFO
FMOD_STUDIO_CPU_USAGE
FMOD_STUDIO_PARAMETER_DESCRIPTION
FMOD_STUDIO_PLUGIN_INSTANCE_PROPERTIES
FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES
FMOD_STUDIO_SOUND_INFO
FMOD_STUDIO_TIMELINE_BEAT_PROPERTIES
FMOD_STUDIO_TIMELINE_MARKER_PROPERTIES
FMOD_STUDIO_USER_PROPERTY
FMOD_STUDIO_ADVANCEDSETTINGS

Settings for advanced features like configuring memory and cpu usage.

C/C++ Syntax

typedef struct {
   int cbsize;
   unsigned int commandqueuesize;
   unsigned int handleinitialsize;
   int studioudateperiod;
   int idlesampleddatapoolsize;
} FMOD_STUDIO_ADVANCEDSETTINGS;

JavaScript Syntax

struct FMOD_STUDIO_ADVANCEDSETTINGS {
   commandqueuesize,
   handleinitialsize,
   studioudateperiod,
   idlesampleddatapoolsize,
};
Members

`cbsize`

[w] Size of this structure. Use `sizeof(FMOD_STUDIO_ADVANCEDSETTINGS)` NOTE: This must be set before calling `Studio::System::getAdvancedSettings` or `Studio::System::setAdvancedSettings`!

`commandqueuesize`

[r/w] Optional. Specify 0 to ignore. Specify the command queue size for studio async processing. Default 32kB.

`handleinitialsize`

[r/w] Optional. Specify 0 to ignore. Specify the initial size to allocate for handles. Memory for handles will grow as needed in pages. Default 8192 * `sizeof(void*)`

`studioupdateperiod`

[r/w] Optional. Specify 0 to ignore. Specify the update period of Studio when in async mode, in milliseconds. Will be quantised to the nearest multiple of mixer duration. Default is 20ms.

`idlesampledatapoolsizes`  

[r/w] Optional. Specify 0 to ignore. Specify the amount of sample data to keep in memory when no longer used, to avoid repeated disk IO. Use -1 to disable. Default is 256kB.
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.
Members marked with [r/w] are either read or write depending on if you are using System::setAdvancedSettings (w) or System::getAdvancedSettings (r).

JavaScript only:

Not all fields are currently supported or may not work as expected at this time.
To initialize an new instance in javascript use "FMOD.STUDIO_ADVANCEDSETTINGS()", no 'new' keyword is required.
See Also

- Studio::System::setAdvancedSettings
- Studio::System::getAdvancedSettings
- FMOD_MODE
Firelight Technologies FMOD Studio API
FMOD_STUDIO_BANK_INFO

Information for loading a bank with Studio::System::loadBankCustom.

C/C++ Syntax

typedef struct {
    int size;
    void *userdata;
    int userdatalength;
    FMOD_FILE_OPEN_CALLBACK opencallback;
    FMOD_FILE_CLOSE_CALLBACK closecallback;
    FMOD_FILE_READ_CALLBACK readcallback;
    FMOD_FILE_SEEK_CALLBACK seekcallback;
} FMOD_STUDIO_BANK_INFO;

JavaScript Syntax

struct FMOD_STUDIO_BANK_INFO
{
    userdata,
    userdatalength,
    opencallback,
    closecallback,
    readcallback,
    seekcallback,
};
Members

size
The size of this struct (for binary compatibility)

userdata
User data to be passed to the file callbacks

userdatalength
If this is non-zero, userdata will be copied internally

opencallback
Callback for opening this file.

closecallback
Callback for closing this file.

readcallback
Callback for reading from this file.

seekcallback
Callback for seeking within this file.
Remarks

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_BANK_INFO()", no 'new' keyword is required.
See Also

- Studio::System::loadBankCustom
Firelight Technologies FMOD Studio API
FMOD_STUDIO_BUFFER_INFO

Information for a single buffer in FMOD Studio.

C/C++ Syntax

typedef struct {
    int currentusage;
    int peakusage;
    int capacity;
    int stallcount;
    float stalltime;
} FMOD_STUDIO_BUFFER_INFO;

JavaScript Syntax

struct FMOD_STUDIO_BUFFER_INFO
{
    currentusage,
    peakusage,
    capacity,
    stallcount,
    stalltime,
};
Members

$currentusage$
Current buffer usage in bytes.

$peakusage$
Peak buffer usage in bytes.

$capacity$
Buffer capacity in bytes.

$stallcount$
Cumulative number of stalls due to buffer overflow.

$stalltime$
Cumulative amount of time stalled due to buffer overflow, in seconds.
Remarks

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_BUFFER_INFO()", no 'new' keyword is required.
See Also

- **FMOD_STUDIO_BUFFER_USAGE**
Firelight Technologies FMOD Studio API
FMOD_STUDIO_BUFFER_USAGE

Information for FMOD Studio buffer usage.

C/C++ Syntax

typedef struct {
    FMOD_STUDIO_BUFFER_INFO studiocommandqueue;
    FMOD_STUDIO_BUFFER_INFO studiohandle;
} FMOD_STUDIO_BUFFER_USAGE;
Members

studiocommandqueue

Information for the Studio Async Command buffer, controlled by
FMOD_STUDIO_ADVANCEDSETTINGS commandQueueSize.

studiohandle

Information for the Studio handle table, controlled by
FMOD_STUDIO_ADVANCEDSETTINGS handleInitialSize.
Remarks

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_BUFFER_USAGE()", no 'new' keyword is required.
See Also

- Studio::System::getBufferUsage
- Studio::System::resetBufferUsage
- FMOD_STUDIO_BUFFER_INFO
Firelight Technologies FMOD Studio API
FMOD_STUDIO_COMMAND_INFO

Information about a single command in a command replay file.

C/C++ Syntax

typedef struct {
    const char *commandname;
    int parentcommandindex;
    int framenumber;
    float frametime;
    FMOD_STUDIO_INSTANCETYPE instancetype;
    FMOD_STUDIO_INSTANCETYPE outputtype;
    unsigned int instancehandle;
    unsigned int outputhandle;
} FMOD_STUDIO_COMMAND_INFO;

JavaScript Syntax

struct FMOD_STUDIO_COMMAND_INFO
{
    commandname,
    parentcommandindex,
    framenumber,
    frametime,
    instancetype,
    outputtype,
    instancehandle,
    outputhandle,
};
Members

`commandname`

The full name of the API function for this command.

`parentcommandindex`

For commands that operate on an instance, this is the command that created the instance.

`framenumber`

The frame the command belongs to.

`frametime`

The playback time at which this command will be executed.

`instancetype`

The type of object that this command uses as an instance.

`outputtype`

The type of object that this command outputs, if any.

`instancehandle`

The original handle value of the instance. This will no longer correspond to any actual object in playback.

`outputhandle`

The original handle value of the command output. This will no longer correspond to any actual object in playback.
Remarks

This information has metadata about the command at the given index. Note that the handle fields are from the recorded session, and will no longer correspond to any actual object type in the current system.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_COMMAND_INFO()", no 'new' keyword is required.
See Also

- Studio::CommandReplay::getCommandInfo
Firelight Technologies FMOD Studio API
FMOD_STUDIO_CPU_USAGE

Performance information for FMOD Studio and low level systems.

C/C++ Syntax

typedef struct {
    float dspusage;
    float streamusage;
    float geometryusage;
    float updateusage;
    float studiousage;
} FMOD_STUDIO_CPU_USAGE;

JavaScript Syntax

struct FMOD_STUDIO_CPU_USAGE
{
    dspusage,
    streamusage,
    geometryusage,
    updateusage,
    studiousage,
};
**Members**

*`dspusage`*

Returns the % CPU time taken by DSP processing on the low level mixer thread.

*`streamusage`*

Returns the % CPU time taken by stream processing on the low level stream thread.

*`geometryusage`*

Returns the % CPU time taken by geometry processing on the low level geometry thread.

*`updateusage`*

Returns the % CPU time taken by low level update, called as part of the studio update.

*`studiousage`*

Returns the % CPU time taken by studio update, called from the studio thread. Does not include low level update time.
Remarks

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_CPU_USAGE()", no 'new' keyword is required.
See Also

- Studio::System::getCPUUsage
Firelight Technologies FMOD Studio API
Structure describing an event parameter.

**C/C++ Syntax**

```c
typedef struct {
    const char *name;
    int index;
    float minimum;
    float maximum;
    float defaultvalue;
    FMOD_STUDIO_PARAMETER_TYPE type;
} FMOD_STUDIO_PARAMETER_DESCRIPTION;
```

**JavaScript Syntax**

```javascript
struct FMOD_STUDIO_PARAMETER_DESCRIPTION {
    name,
    index,
    minimum,
    maximum,
    defaultvalue,
    type,
};
```
**Members**

*name*

Name of the parameter.

*index*

Index of parameter

*minimum*

Minimum parameter value.

*maximum*

Maximum parameter value.

*defaultvalue*

Default value

*type*

Type of the parameter
Remarks

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_PARAMETER_DESCRIPTION()", no 'new' keyword is required.
See Also

- Studio::EventDescription::getParameter
- FMOD_STUDIO_PARAMETER_TYPE
Firelight Technologies FMOD Studio API
FMOD_STUDIO_PLUGIN_INSTANCE_PROPERTIES

This structure holds information about a DSP plugin instance.

C/C++ Syntax

typedef struct {
  const char *name;
  FMOD_DSP *dsp;
} FMOD_STUDIO_PLUGIN_INSTANCE_PROPERTIES;

JavaScript Syntax

struct FMOD_STUDIO_PLUGIN_INSTANCE_PROPERTIES
{
  name,
};
Members

name

The name of the plugin effect or sound (set in FMOD Studio).

dsp

The DSP plugin instance. This can be cast to FMOD::DSP* type.
Remarks

This data is passed to the event callback function when type is `FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_CREATED` or `FMOD_STUDIO_EVENT_CALLBACK_PLUGIN.DestroyED`.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_PLUGIN_INSTANCE_PROPERTIES()", no 'new' keyword is required.
See Also

- `FMOD_STUDIO_EVENT_CALLBACK`
- `Studio::EventDescription::setCallback`
- `Studio::EventInstance::setCallback`
Firelight Technologies FMOD Studio API
This structure holds information about a programmer sound.

### C/C++ Syntax

```c
typedef struct {
    const char *name;
    FMOD_SOUND *sound;
    int subsoundIndex;
} FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES;
```

### JavaScript Syntax

```javascript
struct FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES {
    name, sound, subsoundIndex,
};
```
Members

name

The name of the programmer instrument (set in FMOD Studio).

sound

The programmer-created sound. This should be filled in by the create callback, and cleaned up by the destroy callback. The provided sound should be created with the FMOD_LOOP_NORMAL mode bit set. This can be cast to/from FMOD::Sound* type.

subsoundIndex

The index of the subsound to use. This should be filled in by the create callback, or set to -1 if the provided sound should be used directly. Defaults to -1.
Remarks

This data is passed to the event callback function when type is `FMOD_STUDIO_EVENT_CALLBACK_CREATE_PROGRAMMER_SOUND` or `FMOD_STUDIO_EVENT_CALLBACK_DESTROY_PROGRAMMER_SOUND`.

The provided sound should be created with the `FMOD_LOOP_NORMAL` mode bit set. FMOD will set this bit internally if it is not set, possibly incurring a slight performance penalty.

To support non-blocking loading of FSB subsounds, you can specify the subsound you want to use by setting the subsoundIndex field. This will cause FMOD to wait until the provided sound is ready and then get the specified subsound from it.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_PROGRAMMER_SOUND_PROPERTIES()", no 'new' keyword is required.
See Also

- FMOD_STUDIO_EVENT_CALLBACK
- FMOD_STUDIO_SOUND_INFO
- Studio::EventDescription::setCallback
- Studio::EventInstance::setCallback
- Studio::System::getSoundInfo
Firelight Technologies FMOD Studio API
FMOD_STUDIO_SOUND_INFO

Information for loading a sound from a sound table.

C/C++ Syntax

typedef struct {
    const char *name_or_data;
    FMOD_MODE mode;
    FMOD_CREATE.soundinfo exinfo;
    int subsoundindex;
} FMOD_STUDIO_SOUND_INFO;

JavaScript Syntax

struct FMOD_STUDIO_SOUND_INFO {
    name_or_data,
    mode,
    subsoundindex,
};
Members

name_or_data
The filename or memory buffer that contains the sound.

mode
Mode flags required for loading the sound.

exinfo
Extra information required for loading the sound.

subsoundindex
Subsound index for loading the sound.
Remarks

The name_or_data member points into FMOD internal memory, which will become invalid if the sound table bank is unloaded.

If mode flags such as `FMOD_CREATESTREAM`, `FMOD_CREATECOMPRESSEDSAMPLE` or `FMOD_NONBLOCKING` are required, it is up to the user to OR them together when calling `System::createSound`.

JavaScript only :

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_SOUND_INFO()", no 'new' keyword is required.
See Also

- Studio::System::getSoundInfo
- System::createSound
Firelight Technologies FMOD Studio API
FMOD_STUDIO_TIMELINE_BEAT

This structure holds information about a beat on the timeline.

**C/C++ Syntax**

typedef struct {
    int bar;
    int beat;
    int position;
    float tempo;
    int timesignatureupper;
    int timesignaturelower;
} FMOD_STUDIO_TIMELINE_BEAT_PROPERTIES;

**JavaScript Syntax**

struct FMOD_STUDIO_TIMELINE_BEAT_PROPERTIES {
    bar,
    beat,
    position,
    tempo,
    timesignatureupper,
    timesignaturelower,
};
Members

*bar*
The bar number (starting from 1).

*beat*
The beat number within the bar (starting from 1).

*position*
The position of the beat on the timeline in milliseconds.

*tempo*
The current tempo in beats per minute.

*timesignatureupper*
The current time signature upper number (beats per bar).

*timesignaturelower*
The current time signature lower number (beat unit).
Remarks

This data is passed to the event callback function when type is
FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_BEAT.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use
"FMOD.STUDIO_TIMELINE_BEAT_PROPERTIES()", no 'new' keyword is required.
See Also

- FMOD_STUDIO_EVENT_CALLBACK
- Studio::EventDescription::setCallback
- Studio::EventInstance::setCallback
Firelight Technologies FMOD Studio API
FMOD_STUDIO_TIMELINE_MARKER_PROPERTIES

This structure holds information about a marker on the timeline.

C/C++ Syntax

typedef struct {
    const char *name;
    int position;
} FMOD_STUDIO_TIMELINE_MARKER_PROPERTIES;

JavaScript Syntax

struct FMOD_STUDIO_TIMELINE_MARKER_PROPERTIES {
    name,
    position,
};
Members

*name*

The marker name

*position*

The position of the marker on the timeline in milliseconds.
Remarks

This data is passed to the event callback function when type is
FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_MARKER.

JavaScript only:

Not all fields are currently supported or may not work as expected at this time.
To initialize an new instance in javascript use
"FMOD.STUDIO_TIMELINE_MARKER_PROPERTIES()", no 'new' keyword
is required.
See Also

- **FMOD_STUDIO_EVENT_CALLBACK**
- **Studio::EventDescription::setCallback**
- **Studio::EventInstance::setCallback**
Firelight Technologies FMOD Studio API
**FMOD_STUDIO_USER_PROPERTY**

Structure describing a user property.

**C/C++ Syntax**

```c
typedef struct {
    const char *name;
    FMOD_STUDIO_USER_PROPERTY_TYPE type;
    int intvalue;
    FMOD_BOOL boolvalue;
    float floatvalue;
    const char *stringvalue;
} FMOD_STUDIO_USER_PROPERTY;
```

**JavaScript Syntax**

```javascript
struct FMOD_STUDIO_USER_PROPERTY {
    name,
    type,
    intvalue,
    boolvalue,
    floatvalue,
    stringvalue,
};
```
Members

name

Name of the user property.

type

Type of the user property. Use this to select one of the following values.

intvalue

Value of the user property. Only valid when type is
FMOD_STUDIO_USER_PROPERTY_TYPE.INTEGER.

boolvalue

Value of the user property. Only valid when type is
FMOD_STUDIO_USER_PROPERTY_TYPE.BOOLEAN.

floatvalue

Value of the user property. Only valid when type is
FMOD_STUDIO_USER_PROPERTY_TYPE.FLOAT.

stringvalue

Value of the user property. Only valid when type is
FMOD_STUDIO_USER_PROPERTY_TYPE.STRING.
Remarks

JavaScript only:

Not all fields are currently supported or may not work as expected at this time. To initialize an new instance in javascript use "FMOD.STUDIO_USERPROPERTY()", no 'new' keyword is required.
See Also

- Studio::EventDescription::getUserProperty
Firelight Technologies FMOD Studio API
Defines

FMOD_STUDIO_COMMANDCAPTURE_FLAGS
FMOD_STUDIO_COMMANDREPLAY_FLAGS
FMOD_STUDIO_EVENT_CALLBACK_TYPE
FMOD_STUDIO_INITFLAGS
FMOD_STUDIO_LOAD_BANK_FLAGS
FMOD_STUDIO_LOAD_MEMORY_ALIGNMENT
FMOD_STUDIO_SYSTEM_CALLBACK_TYPE
Firelight Technologies FMOD Studio API
Flags passed into `Studio::System::startCommandCapture`.

C/C++ Syntax

```c
#define FMOD_STUDIO_COMMANDCAPTURE_NORMAL 0x00000000
#define FMOD_STUDIO_COMMANDCAPTURE_FILEFLUSH 0x00000001
#define FMOD_STUDIO_COMMANDCAPTURE_SKIP_INITIAL_STATE 0x00000002
```

JavaScript Syntax

```javascript
FMOD.STUDIO_COMMANDCAPTURE_NORMAL
FMOD.STUDIO_COMMANDCAPTURE_FILEFLUSH
FMOD.STUDIO_COMMANDCAPTURE_SKIP_INITIAL_STATE
```
Values

**FMOD_STUDIO_COMMANDCAPTURE_NORMAL**

Standard behaviour.

**FMOD_STUDIO_COMMANDCAPTURE_FILEFLUSH**

Call file flush on every command.

**FMOD_STUDIO_COMMANDCAPTURE_SKIP_INITIAL_STATE**

Normally the initial state of banks and instances is captured, unless this flag is set.
See Also

- Studio::System::startCommandCapture
Firelight Technologies FMOD Studio API
Flags passed into `Studio::System::loadCommandReplay`.

C/C++ Syntax

```c
#define FMOD_STUDIO_COMMANDREPLAY_NORMAL 0x00000000
#define FMOD_STUDIO_COMMANDREPLAY_SKIP_CLEANUP 0x00000001
#define FMOD_STUDIO_COMMANDREPLAY_FAST_FORWARD 0x00000002
```

JavaScript Syntax

```javascript
FMOD.STUDIO_COMMANDREPLAY_NORMAL
FMOD.STUDIO_COMMANDREPLAY_SKIP_CLEANUP
FMOD.STUDIO_COMMANDREPLAY_FAST_FORWARD
```
Values

*FMOD\_STUDIO\_COMMANDREPLAY\_NORMAL*

Standard behaviour.

*FMOD\_STUDIO\_COMMANDREPLAY\_SKIP\_CLEANUP*

Normally the playback will release any created resources when it stops, unless this flag is set.

*FMOD\_STUDIO\_COMMANDREPLAY\_FAST\_FORWARD*

Play back at maximum speed, ignoring the timing of the original replay.
See Also

- Studio::System::loadCommandReplay
Firelight Technologies FMOD Studio API
These callback types are used with `FMOD_STUDIO_EVENT_CALLBACK).

The data passed to the event callback function in the `parameters` argument varies based on the callback type.

**FMOD_STUDIO_EVENT_CALLBACK_STARTING** is called when:

- `Studio::EventInstance::start` has been called on an event which was not already playing. The event will remain in this state until its sample data has been loaded. If the event could not be started due to polyphony, then `FMOD_STUDIO_EVENT_CALLBACK_START_FAILED` will be called instead.

**FMOD_STUDIO_EVENT_CALLBACK_STARTED** is called when:

- The event has commenced playing. Normally this callback will be issued immediately after `FMOD_STUDIO_EVENT_CALLBACK_STARTING`, but may be delayed until sample data has loaded.

**FMOD_STUDIO_EVENT_CALLBACK_RESTARTED** is called when:

- `Studio::EventInstance::start` has been called on an event which was already playing.

**FMOD_STUDIO_EVENT_CALLBACK_STOPPED** is called when:

- The event has stopped due to `Studio::EventInstance::stop` being called with `FMOD_STUDIO_STOP_IMMEDIATE`.
- The event has finished fading out after `Studio::EventInstance::stop` was called with `FMOD_STUDIO_STOP_ALLOWFADEOUT`.
- The event has stopped naturally by reaching the end of the timeline, and no further sounds can be triggered due to parameter changes.

**FMOD_STUDIO_EVENT_CALLBACK_START_FAILED** is called when:

- `Studio::EventInstance::start` has been called but the polyphony settings did
not allow the event to start. In this case none of
FMOD_STUDIO_EVENT_CALLBACK_STARTING,
FMOD_STUDIO_EVENT_CALLBACK_STARTED and
FMOD_STUDIO_EVENT_CALLBACK_STOPPED will not be called.

**FMOD_STUDIO_EVENT_CALLBACK_CREATE_PROGRAMMER_SOUND**
is called when:

- A programmer sound is about to play. FMOD expects the callback to provide an FMOD::Sound object for it to use.

**FMOD_STUDIO_EVENT_CALLBACK_DESTROY_PROGRAMMER_SOUND**
is called when:

- A programmer sound has stopped playing. At this point it is safe to release the FMOD::Sound object that was used.

**C/C++ Syntax**

```c
#define FMOD_STUDIO_EVENT_CALLBACK_CREATED 0x00000001
#define FMOD_STUDIO_EVENT_CALLBACK_DESTROYED 0x00000002
#define FMOD_STUDIO_EVENT_CALLBACK_STARTING 0x00000004
#define FMOD_STUDIO_EVENT_CALLBACK_STARTED 0x00000008
#define FMOD_STUDIO_EVENT_CALLBACK_RESTARTED 0x00000010
#define FMOD_STUDIO_EVENT_CALLBACK_STOPPED 0x00000020
#define FMOD_STUDIO_EVENT_CALLBACK_START_FAILED 0x00000040
#define FMOD_STUDIO_EVENT_CALLBACK_CREATE_PROGRAMMER_SOUND 0x00000080
#define FMOD_STUDIO_EVENT_CALLBACK_DESTROY_PROGRAMMER_SOUND 0x00000100
#define FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_CREATED 0x00000200
#define FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_DESTROYED 0x00000400
#define FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_MARKER 0x00000800
#define FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_BEAT 0x00001000
#define FMOD_STUDIO_EVENT_CALLBACK_SOUND_PLAYED 0x00002000
#define FMOD_STUDIO_EVENT_CALLBACK_SOUND_STOPPED 0x00004000
#define FMOD_STUDIO_EVENT_CALLBACK_ALL 0xFFFFFFFF
```

**JavaScript Syntax**

```javascript
FMOD.STUDIO_EVENT_CALLBACK_CREATED
FMOD.STUDIO_EVENT_CALLBACK_DESTROYED
FMOD.STUDIO_EVENT_CALLBACK_STARTING
FMOD.STUDIO_EVENT_CALLBACK_STARTED
FMOD.STUDIO_EVENT_CALLBACK_RESTARTED
FMOD.STUDIO_EVENT_CALLBACK_STOPPED
```
FMOD.STUDIO_EVENT_CALLBACK_START_FAILED
FMOD.STUDIO_EVENT_CALLBACK_CREATE_PROGRAMMER_SOUND
FMOD.STUDIO_EVENT_CALLBACK_DESTROY_PROGRAMMER_SOUND
FMOD.STUDIO_EVENT_CALLBACK_PLUGIN_CREATED
FMOD.STUDIO_EVENT_CALLBACK_PLUGIN_DESTROYED
FMOD.STUDIO_EVENT_CALLBACK_TIMELINE_MARKER
FMOD.STUDIO_EVENT_CALLBACK_TIMELINE_BEAT
FMOD.STUDIO_EVENT_CALLBACK_SOUND_PLAYED
FMOD.STUDIO_EVENT_CALLBACK_SOUND_STOPPED
FMOD.STUDIO_EVENT_CALLBACK_ALL
Values

`FMOD_STUDIO_EVENT_CALLBACK_CREATED`
Called when an instance is fully created. Parameters = unused.

`FMOD_STUDIO_EVENT_CALLBACK_DESTROYED`
Called when an instance is just about to be destroyed. Parameters = unused.

`FMOD_STUDIO_EVENT_CALLBACK_STARTING`
Called when an instance is preparing to start. Parameters = unused.

`FMOD_STUDIO_EVENT_CALLBACK_STARTED`
Called when an instance starts playing. Parameters = unused.

`FMOD_STUDIO_EVENT_CALLBACK_RESTARTED`
Called when an instance is restarted. Parameters = unused.

`FMOD_STUDIO_EVENT_CALLBACK_STOPPED`
Called when an instance stops. Parameters = unused.

`FMOD_STUDIO_EVENT_CALLBACK_START_FAILED`
Called when an instance did not start, e.g. due to polyphony. Parameters = unused.

`FMOD_STUDIO_EVENT_CALLBACK_CREATE_PROGRAMMER_SOUND`
Called when a programmer sound needs to be created in order to play a programmer instrument. Parameters =

`FMOD_STUDIO_EVENT_CALLBACK_DESTROY_PROGRAMMER_SOUND`
Called when a programmer sound needs to be destroyed. Parameters = FMOD_STUDIO_PROGRAMMER_SOUND_PROPERTIES.

**FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_CREATED**

Called when a DSP plugin instance has just been created. Parameters = FMOD_STUDIO_PLUGIN_INSTANCE_PROPERTIES.

**FMOD_STUDIO_EVENT_CALLBACK_PLUGIN_DESTROYED**

Called when a DSP plugin instance is about to be destroyed. Parameters = FMOD_STUDIO_PLUGIN_INSTANCE_PROPERTIES.

**FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_MARKER**

Called when the timeline passes a named marker. Parameters = FMOD_STUDIO_TIMELINE_MARKER_PROPERTIES.

**FMOD_STUDIO_EVENT_CALLBACK_TIMELINE_BEAT**

Called when the timeline hits a beat in a tempo section. Parameters = FMOD_STUDIO_TIMELINE_BEAT_PROPERTIES.

**FMOD_STUDIO_EVENT_CALLBACK_SOUND_PLAYED**

Called when the event plays a sound. Parameters = FMOD::Sound.

**FMOD_STUDIO_EVENT_CALLBACK_SOUND_STOPPED**

Called when the event finishes playing a sound. Parameters = FMOD::Sound.

**FMOD_STUDIO_EVENT_CALLBACK_ALL**

Pass this mask to Studio::EventDescription::setCallback or Studio::EventInstance::setCallback to receive all callback types.
See Also

- Studio::EventDescription::setCallback
- Studio::EventInstance::setCallback
- FMOD_STUDIO_EVENT_CALLBACK
Firelight Technologies FMOD Studio API
FMOD_STUDIO_INITFLAGS

Studio System initialization flags. Use them with `Studio::System::initialize` in the `studioflags` parameter to change various behavior.

**C/C++ Syntax**

```c
#define FMOD_STUDIO_INIT_NORMAL 0x00000000
#define FMOD_STUDIO_INIT_LIVEUPDATE 0x00000001
#define FMOD_STUDIO_INIT_ALLOW_MISSING_PLUGINS 0x00000002
#define FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE 0x00000004
#define FMOD_STUDIO_INIT_DEFERRED_CALLBACKS 0x00000008
#define FMOD_STUDIO_INIT_LOAD_FROM_UPDATE 0x00000010
```

**JavaScript Syntax**

```javascript
FMOD.STUDIO_INIT_NORMAL
FMOD.STUDIO_INIT_LIVEUPDATE
FMOD.STUDIO_INIT_ALLOW_MISSING_PLUGINS
FMOD.STUDIO_INIT_SYNCHRONOUS_UPDATE
FMOD.STUDIO_INIT_DEFERRED_CALLBACKS
FMOD.STUDIO_INIT_LOAD_FROM_UPDATE
```
Values

**FMOD_STUDIO_INIT_NORMAL**

Initialize normally.

**FMOD_STUDIO_INIT_LIVEUPDATE**

Enable live update.

**FMOD_STUDIO_INIT_ALLOW_MISSING_PLUGINS**

Load banks even if they reference plugins that have not been loaded.

**FMOD_STUDIO_INIT_SYNCHRONOUS_UPDATE**

Disable asynchronous processing and perform all processing on the calling thread instead.

**FMOD_STUDIO_INIT_DEFERRED_CALLBACKS**

Defer timeline callbacks until the main update. See Studio::EventInstance::setCallback for more information.

**FMOD_STUDIO_INIT_LOAD_FROM_UPDATE**

No additional threads are created for bank and resource loading. Loading is driven from Studio::System::update. Mainly used in non-realtime situations.
See Also

- Studio::System::initialize
Firelight Technologies FMOD Studio API
FMOD_STUDIO_LOAD_BANK_FLAGS

Flags passed into Studio loadBank commands to control bank load behaviour.

C/C++ Syntax

```c
#define FMOD_STUDIO_LOAD_BANK_NORMAL 0x00000000
#define FMOD_STUDIO_LOAD_BANK_NONBLOCKING 0x00000001
#define FMOD_STUDIO_LOAD_BANK_DECOMPRESS_SAMPLES 0x00000002
```

JavaScript Syntax

```javascript
FMOD.STUDIO_LOAD_BANK_NORMAL
FMOD.STUDIO_LOAD_BANK_NONBLOCKING
FMOD.STUDIO_LOAD_BANK_DECOMPRESS_SAMPLES
```
Values

**FMOD_STUDIO_LOAD_BANK_NORMAL**

Standard behaviour.

**FMOD_STUDIO_LOAD_BANK_NONBLOCKING**

Bank loading occurs asynchronously rather than occurring immediately.

**FMOD_STUDIO_LOAD_BANK_DECOMPRESS_SAMPLES**

Force samples to decompress into memory when they are loaded, rather than staying compressed.
See Also

- Studio::System::loadBankFile
- Studio::System::loadBankMemory
- Studio::System::loadBankCustom

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
The required alignment of the buffer for `Studio::System::loadBankMemory` when using `FMOD_STUDIO_LOAD_MEMORY_POINT`.

**C/C++ Syntax**

```c
#define FMOD_STUDIO_LOAD_MEMORY_ALIGNMENT 32
```

**JavaScript Syntax**

```javascript
FMOD.STUDIO_LOAD_MEMORY_ALIGNMENT
```
Values

FMOD_STUDIO_LOAD_MEMORY_ALIGNMENT
See Also

- Studio::System::loadBankMemory
- FMOD_STUDIO_LOAD_MEMORY_MODE
Firelight Technologies FMOD Studio API
FMOD_STUDIO_SYSTEM_CALLBACK

These callback types are used with `Studio::System::setCallback`.

**C/C++ Syntax**

```c
#define FMOD_STUDIO_SYSTEM_CALLBACK_PREUPDATE 0x00000001
#define FMOD_STUDIO_SYSTEM_CALLBACK_POSTUPDATE 0x00000002
#define FMOD_STUDIO_SYSTEM_CALLBACK_BANK_UNLOAD 0x00000004
#define FMOD_STUDIO_SYSTEM_CALLBACK_ALL 0xFFFFFFFF
```

**JavaScript Syntax**

```javascript
FMOD.STUDIO_SYSTEM_CALLBACK_PREUPDATE
FMOD.STUDIO_SYSTEM_CALLBACK_POSTUPDATE
FMOD.STUDIO_SYSTEM_CALLBACK_BANK_UNLOAD
FMOD.STUDIO_SYSTEM_CALLBACK_ALL
```
Values

*FMOD_STUDIO_SYSTEM_CALLBACK_PREUPDATE*

Called at the start of the main Studio update. For async mode this will be on its own thread.

*FMOD_STUDIO_SYSTEM_CALLBACK_POSTUPDATE*

Called at the end of the main Studio update. For async mode this will be on its own thread.

*FMOD_STUDIO_SYSTEM_CALLBACK_BANK_UNLOAD*

Called when bank has just been unloaded, after all resources are freed. CommandData will be the bank handle.

*FMOD_STUDIO_SYSTEM_CALLBACK_ALL*

Pass this mask to `Studio::System::setCallback` to receive all callback types.
See Also

- `FMOD_STUDIO_SYSTEM_CALLBACK`
- `Studio::System::setCallback`
Firelight Technologies FMOD Studio API
Enumerations

FMOD_STUDIO_EVENT_PROPERTY
FMOD_STUDIO_INSTANCETYPE
FMOD_STUDIO>Loading_STATE
FMOD_STUDIO_LOAD_MEMORY_MODE
FMOD_STUDIO_PARAMETER_TYPE
FMOD_STUDIO_PLAYBACK_STATE
FMOD_STUDIO_STOP_MODE
FMOD_STUDIO_USERPROPERTY_TYPE
Firelight Technologies FMOD Studio API
These definitions describe built-in event properties.

C/C++ Syntax

typedef enum {
    FMOD_STUDIO_EVENT_PROPERTY_CHANNELPRIORITY,
    FMOD_STUDIO_EVENT_PROPERTY_SCHEDULE_DELAY,
    FMOD_STUDIO_EVENT_PROPERTY_SCHEDULE_LOOKAHEAD,
    FMOD_STUDIO_EVENT_PROPERTY_MINIMUM_DISTANCE,
    FMOD_STUDIO_EVENT_PROPERTY_MAXIMUM_DISTANCE,
    FMOD_STUDIO_EVENT_PROPERTY_MAX
} FMOD_STUDIO_EVENT_PROPERTY;

JavaScript Syntax

FMOD.STUDIO_EVENT_PROPERTY_CHANNELPRIORITY
FMOD.STUDIO_EVENT_PROPERTY_SCHEDULE_DELAY
FMOD.STUDIO_EVENT_PROPERTY_SCHEDULE_LOOKAHEAD
FMOD.STUDIO_EVENT_PROPERTY_MINIMUM_DISTANCE
FMOD.STUDIO_EVENT_PROPERTY_MAXIMUM_DISTANCE
FMOD.STUDIO_EVENT_PROPERTY_MAX
FMOD.STUDIO_EVENT_PROPERTY_FORCEINT
Values

**FMOD_STUDIO_EVENT_PROPERTY_CHANNEL_PRIORITY**
Priority to set on low-level channels created by this event instance (-1 to 256).

**FMOD_STUDIO_EVENT_PROPERTY_SCHEDULE_DELAY**
Schedule delay to synchronized playback for multiple tracks in DSP clocks, or -1 for default.

**FMOD_STUDIO_EVENT_PROPERTY_SCHEDULE_LOOKAHEAD**
Schedule look-ahead on the timeline in DSP clocks, or -1 for default.

**FMOD_STUDIO_EVENT_PROPERTY_MINIMUM_DISTANCE**
Override the event's 3D minimum distance, or -1 for default.

**FMOD_STUDIO_EVENT_PROPERTY_MAXIMUM_DISTANCE**
Override the event's 3D maximum distance, or -1 for default.

**FMOD_STUDIO_EVENT_PROPERTY_MAX**
Maximum number of event properties supported.
Remarks

For `FMOD_STUDIO_EVENTPROPERTY_CHANNELPRIORITY`, a value of -1 uses the priority set in FMOD Studio, while other values override it. This property uses the same system as `Channel::setPriority`; this means lower values are higher priority (i.e. 0 is the highest priority while 256 is the lowest).
See Also

- Studio::EventInstance::getProperty
- Studio::EventInstance::setProperty

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
FMOD_STUDIO_INSTANCECETYPE

Used to distinguish the types used in command replays.

C/C++ Syntax

typedef enum {
    FMOD_STUDIO_INSTANCECETYPE_NONE,
    FMOD_STUDIO_INSTANCECETYPE_SYSTEM,
    FMOD_STUDIO_INSTANCECETYPE_EVENTDESCRIPTION,
    FMOD_STUDIO_INSTANCECETYPE_EVENTINSTANCE,
    FMOD_STUDIO_INSTANCECETYPE_PARAMETERINSTANCE,
    FMOD_STUDIO_INSTANCECETYPE_BUS,
    FMOD_STUDIO_INSTANCECETYPE_VCA,
    FMOD_STUDIO_INSTANCECETYPE_BANK,
    FMOD_STUDIO_INSTANCECETYPE_COMMANDREPLAY
} FMOD_STUDIO_INSTANCECETYPE;

JavaScript Syntax

FMOD.STUDIO_INSTANCECETYPE_NONE
FMOD.STUDIO_INSTANCECETYPE_SYSTEM
FMOD.STUDIO_INSTANCECETYPE_EVENTDESCRIPTION
FMOD.STUDIO_INSTANCECETYPE_EVENTINSTANCE
FMOD.STUDIO_INSTANCECETYPE_PARAMETERINSTANCE
FMOD.STUDIO_INSTANCECETYPE_BUS
FMOD.STUDIO_INSTANCECETYPE_VCA
FMOD.STUDIO_INSTANCECETYPE_BANK
FMOD.STUDIO_INSTANCECETYPE_COMMANDREPLAY
FMOD.STUDIO_INSTANCECETYPE_FORCEINT
Values

FMOD_STUDIO_INSTANCETYPE_NONE
FMOD_STUDIO_INSTANCETYPE_SYSTEM
FMOD_STUDIO_INSTANCETYPE_EVENTDESCRIPTION
FMOD_STUDIO_INSTANCETYPE_EVENTINSTANCE
FMOD_STUDIO_INSTANCETYPE_PARAMETERINSTANCE
FMOD_STUDIO_INSTANCETYPE_BUS
FMOD_STUDIO_INSTANCETYPE_VCA
FMOD_STUDIO_INSTANCETYPE_BANK
FMOD_STUDIO_INSTANCETYPE_COMMANDREPLAY
Firelight Technologies FMOD Studio API
FMOD_STUDIO_LOADING_STATE

These values describe the loading status of various objects.

C/C++ Syntax

typedef enum {
    FMOD_STUDIO_LOADING_STATE_UNLOADING,
    FMOD_STUDIO_LOADING_STATE_UNLOADED,
    FMOD_STUDIO_LOADING_STATE_LOADING,
    FMOD_STUDIO_LOADING_STATE_LOADED,
    FMOD_STUDIO_LOADING_STATE_ERROR
} FMOD_STUDIO_LOADING_STATE;

JavaScript Syntax

FMOD.STUDIO_LOADING_STATE_UNLOADING
FMOD.STUDIO_LOADING_STATE_UNLOADED
FMOD.STUDIO_LOADING_STATE_LOADING
FMOD.STUDIO_LOADING_STATE_LOADED
FMOD.STUDIO_LOADING_STATE_ERROR
FMOD.STUDIO_LOADING_STATE_FORCEINT
**Values**

*FMOD_STUDIO_LOADING_STATE_UNLOADING*

Currently unloading.

*FMOD_STUDIO_LOADING_STATE_UNLOADED*

Not loaded.

*FMOD_STUDIO_LOADING_STATE_LOADING*

Loading in progress.

*FMOD_STUDIO_LOADING_STATE_LOADED*

Loaded and ready to play.

*FMOD_STUDIO_LOADING_STATE_ERROR*

Failed to load and is now in error state.
Remarks

 Calling `Studio::System::loadBankFile`, `Studio::System::loadBankMemory` or `Studio::System::loadBankCustom` will trigger loading of metadata from the bank.

 Calling `Studio::EventDescription::loadSampleData`, `Studio::EventDescription::createInstance` or `Studio::Bank::loadSampleData` may trigger asynchronous loading of sample data.
See Also

- Studio::EventDescription::getSampleLoadingState
- Studio::Bank::getLoadingState
- Studio::Bank::getSampleLoadingState

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
FMOD_STUDIO_LOAD_MEMORY

Specifies how to use the memory buffer passed to Studio::System::loadBankMemory.

C/C++ Syntax

typedef enum {
    FMOD_STUDIO_LOAD_MEMORY,
    FMOD_STUDIO_LOAD_MEMORY_POINT
} FMOD_STUDIO_LOAD_MEMORY_MODE;

JavaScript Syntax

FMOD.STUDIO_LOAD_MEMORY
FMOD.STUDIO_LOAD_MEMORY_POINT
FMOD.STUDIO_LOAD_MEMORY_FORCEINT
Values

*FMOD_STUDIO_LOAD_MEMORY*

Duplicates the memory into its own buffers, memory can be freed after
`Studio::System::loadBankMemory` returns.

*FMOD_STUDIO_LOAD_MEMORY_POINT*

Copies the memory pointer without duplicating the memory into its own buffers, memory can be freed after receiving a
`FMOD_STUDIO_SYSTEM_CALLBACK_BANK_UNLOAD` callback.
See Also

- `Studio::System::loadBankMemory`
- `Studio::Bank::unload`
Firelight Technologies FMOD Studio API
FMOD_STUDIO_PARAMETER_TYPE

Describes the type of a parameter.

C/C++ Syntax

typedef enum {
    FMOD_STUDIO_PARAMETER_GAME_CONTROLLED,
    FMOD_STUDIO_PARAMETER_AUTOMATIC_DISTANCE,
    FMOD_STUDIO_PARAMETER_AUTOMATIC_EVENT_CONE_ANGLE,
    FMOD_STUDIO_PARAMETER_AUTOMATIC_EVENT_ORIENTATION,
    FMOD_STUDIO_PARAMETER_AUTOMATIC_DIRECTION,
    FMOD_STUDIO_PARAMETER_AUTOMATIC_ELEVATION,
    FMOD_STUDIO_PARAMETER_AUTOMATIC_LISTENER_ORIENTATION,
    FMOD_STUDIO_PARAMETER_MAX
} FMOD_STUDIO_PARAMETER_TYPE;

JavaScript Syntax

FMOD.STUDIO_PARAMETER_GAME_CONTROLLED
FMOD.STUDIO_PARAMETER_AUTOMATIC_DISTANCE
FMOD.STUDIO_PARAMETER_AUTOMATIC_EVENT_CONE_ANGLE
FMOD.STUDIO_PARAMETER_AUTOMATIC_EVENT_ORIENTATION
FMOD.STUDIO_PARAMETER_AUTOMATIC_DIRECTION
FMOD.STUDIO_PARAMETER_AUTOMATIC_ELEVATION
FMOD.STUDIO_PARAMETER_AUTOMATIC_LISTENER_ORIENTATION
FMOD.STUDIO_PARAMETER_MAX
FMOD.STUDIO_PARAMETER_FORCEINT
Values

FMOD_STUDIO_PARAMETER_GAME_CONTROLLED

Controlled via the API using Studio::EventInstance::setParameterValue.

FMOD_STUDIO_PARAMETER_AUTOMATIC_DISTANCE

Distance between the event and the listener.

FMOD_STUDIO_PARAMETER_AUTOMATIC_EVENT_CONE_ANGLE

Angle between the event's forward vector and the vector pointing from the event to the listener (0 to 180 degrees).

FMOD_STUDIO_PARAMETER_AUTOMATIC_EVENT_ORIENTATION

Horizontal angle between the event's forward vector and listener's forward vector (-180 to 180 degrees).

FMOD_STUDIO_PARAMETER_AUTOMATIC_DIRECTION

Horizontal angle between the listener's forward vector and the vector pointing from the listener to the event (-180 to 180 degrees).

FMOD_STUDIO_PARAMETER_AUTOMATIC_ELEVATION

Angle between the listener's XZ plane and the vector pointing from the listener to the event (-90 to 90 degrees).

FMOD_STUDIO_PARAMETER_AUTOMATIC_LISTENER_ORIENTATION

Horizontal angle between the listener's forward vector and the global positive Z axis (-180 to 180 degrees).

FMOD_STUDIO_PARAMETER_MAX

Maximum number of parameter types supported.
Remarks

There are two primary types of parameters: game controlled and automatic. Game controlled parameters receive their value from the API using `Studio::EventInstance::setParameterValue`. Automatic parameters are updated inside FMOD based on the positional information of the event and listener.

**Horizontal angle** means the angle between vectors projected onto the listener's XZ plane (for the EVENT_ORIENTATION and DIRECTION parameters) or the global XZ plane (for the LISTENER_ORIENTATION parameter).
See Also

- **FMOD_STUDIO_PARAMETER_DESCRIPTION**
- **Studio::EventInstance::setParameterValue**
- **Studio::EventInstance::set3DAttributes**
- **Studio::System::setListenerAttributes**
FMOD_STUDIO_PLAYBACK_STATE

These values describe the playback state of an event instance.

**C/C++ Syntax**

typedef enum {
    FMOD_STUDIO_PLAYBACK_PLAYING,
    FMOD_STUDIO_PLAYBACK_SUSTAINING,
    FMOD_STUDIO_PLAYBACK_STOPPED,
    FMOD_STUDIO_PLAYBACK_STARTING,
    FMOD_STUDIO_PLAYBACK_STOPPING
} FMOD_STUDIO_PLAYBACK_STATE;

**JavaScript Syntax**

FMOD.STUDIO_PLAYBACK_PLAYING
FMOD.STUDIO_PLAYBACK_SUSTAINING
FMOD.STUDIO_PLAYBACK_STOPPED
FMOD.STUDIO_PLAYBACK_STARTING
FMOD.STUDIO_PLAYBACK_STOPPING
FMOD.STUDIO_PLAYBACK_FORCEINT
**Values**

*FMOD_STUDIO_PLAYBACK_PLAYING*

Currently playing.

*FMOD_STUDIO_PLAYBACK_SUSTAINING*

The timeline cursor is paused on a sustain point.

*FMOD_STUDIO_PLAYBACK_STOPPED*

Not playing.

*FMOD_STUDIO_PLAYBACK_STARTING*

Start has been called but the instance is not fully started yet.

*FMOD_STUDIO_PLAYBACK_STOPPING*

Stop has been called but the instance is not fully stopped yet.
See Also

- `Studio::EventInstance::getPlaybackState`
- `Studio::EventInstance::start`
- `Studio::EventInstance::stop`
- `FMOD_STUDIO_EVENT_CALLBACK_TYPE`
Firelight Technologies FMOD Studio API
FMOD_STUDIO_STOP_MODE

Controls how to stop playback of an event instance.

C/C++ Syntax

typedef enum {
    FMOD_STUDIO_STOP_ALLOWFADEOUT,
    FMOD_STUDIO_STOP_IMMEDIATE
} FMOD_STUDIO_STOP_MODE;

JavaScript Syntax

FMOD.STUDIO_STOP_ALLOWFADEOUT
FMOD.STUDIO_STOP_IMMEDIATE
FMOD.STUDIO_STOP_FORCEINT
Values

`FMOD_STUDIO_STOP_ALLOWFADEOUT`

Allows AHDSR modulators to complete their release, and DSP effect tails to play out.

`FMOD_STUDIO_STOP_IMMEDIATE`

Stops the event instance immediately.
See Also

- Studio::EventInstance::stop
- Studio::Bus::stopAllEvents
Firelight Technologies FMOD Studio API
These definitions describe a user property's type.

**C/C++ Syntax**

```c
typedef enum {
    FMOD_STUDIO_USER_PROPERTY_TYPE_INTEGER,
    FMOD_STUDIO_USER_PROPERTY_TYPE_BOOLEAN,
    FMOD_STUDIO_USERPROPERTYTYPEFLOAT,
    FMOD_STUDIO_USERPROPERTYTYPESTRING
} FMOD_STUDIO_USER_PROPERTY_TYPE;
```

**JavaScript Syntax**

```javascript
FMOD.STUDIO_USER_PROPERTY_TYPE_INTEGER
FMOD.STUDIO_USER_PROPERTY_TYPE_BOOLEAN
FMOD.STUDIO_USER_PROPERTY_TYPE_FLOAT
FMOD.STUDIO_USER_PROPERTY_TYPE_STRING
FMOD.STUDIO_USER_PROPERTY_TYPE_FORCEINT
```
Values

*FMOD_STUDIO_USER_PROPERTY_TYPE_INTEGER*

Integer property

*FMOD_STUDIO_USER_PROPERTY_TYPE_BOOLEAN*

Boolean property

*FMOD_STUDIO_USER_PROPERTY_TYPE_FLOAT*

Float property

*FMOD_STUDIO_USER_PROPERTY_TYPE_STRING*

String property
See Also

- FMOD_STUDIO_USER_PROPERTY
Firelight Technologies FMOD Studio API
FSBank API

Functions  Callbacks
Structures
Defines
Enumerations
Firelight Technologies FMOD Studio API
Functions

FSBank_Build  FSBank_BuildCancel
FSBank_FetchFSBMemory
FSBank_FetchNextProgressItem
FSBank_Init
FSBank_MemoryGetStats
FSBank_MemoryInit
FSBank_Release
FSBank_ReleaseProgressItem
Firelight Technologies FMOD Studio API
FSBank_Build

Begin the building process for the provided subsound descriptions, function will block until complete.

C Syntax

FSBANK_RESULT FSBank_Build(
    const FSBANK_SUBSOUND *subSounds,
    unsigned int numSubSounds,
    FSBANK_FORMAT encodeFormat,
    FSBANK_BUILDFLAGS buildFlags,
    unsigned int quality,
    const char *encryptKey,
    const char *outputFileName
);
Parameters

$subSounds$

An array of subsound descriptions each defining one subsound for the final FSB.

$numSubSounds$

The number of elements in the 'subSounds' array.

$encodeFormat$

The format to be used for encoding the FSB.

$buildFlags$

Building flags which control how the sample data is encoded.

$quality$

Controls the quality level after compression. From 1 (high compression / low quality) to 100 (high quality / low compression), use 0 for default quality. See remarks for format specific usage.

$encryptKey$

Optional string 'key' used to encrypt the FSB, same key is required at runtime for decryption.

$outputFileName$

Name (and path) of the FSB to produce.
Return Values

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

- **AT9** - Bitrate (Kbps) depends on channel count, quality [1 to 100] maps linearly to the available options
  - 1ch = [36, 48, 60, 72, 84, 96]
  - 2ch = [72, 96, 120, 144, 168, 192]
- **MPEG** - Bitrate (Kbps) = FMOD quality * 3.2
- **Vorbis** - Vorbis quality [-0.1 to 1.0] maps linearly to FMOD quality [1 to 100]
- **XMA** - XMA quality = FMOD quality
See Also

- FSBANK_SUBSOUND
- FSBANK_FORMAT
- FSBANK_BUILDFLAGS
Firelight Technologies FMOD Studio API
**FSBank_BuildCancel**

Halt the build in progress, must be called from a different thread to **FSBank_Build**.

**C Syntax**

```c
FSBANK_RESULT FSBank_BuildCancel();
```
Return Values

If the function succeeds then the return value is `FSBANK_OK`. If the function fails then the return value will be one of the values defined in the `FSBANK_RESULT` enumeration.
See Also

- FSBank_Build
Firelight Technologies FMOD Studio API
FSBank_FetchFSBMemory

After FSBank_Build has completed, fetch the built FSB data from memory.

C Syntax

FSBANK_RESULT FSBank_FetchFSBMemory(
    const void **data,
    unsigned int *length
);
Parameters

\textit{data}

Address of a variable that receives a pointer to the built FSB data.

\textit{length}

Address of a variable that receives the length of the FSB data pointed to by 'data'.
Return Values

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

Only available if outputFileName is set to NULL during FSBank_Build.

The memory allocated as part of FSBank_Build will be freed automatically by the next FSBank_Build or FSBank_Release.
See Also

- FSBank_Build
- FSBank_Release

Version 1.10.02 Built on Dec 7, 2017
Firelight Technologies FMOD Studio API
**FSBank_FetchNextProgressItem**

Fetch build progress items that describe the current state of the build. Can be called while the build is in progress to get realtime updates or after the build for a report. Call **FSBank_ReleaseProgressItem** to free allocated memory.

**C Syntax**

```c
FSBANK_RESULT FSBank_FetchNextProgressItem(
    const FSBANK_PROGRESSITEM **progressItem
);
```
Parameters

*progressItem*

One status update about the progress of a particular subsound.
Return Values

If the function succeeds then the return value is **FSBANK_OK**.
If the function fails then the return value will be one of the values defined in the
**FSBANK_RESULT** enumeration.
See Also

- FSBank_ReleaseProgressItem
- FSBANK_PROGRESSITEM
Firelight Technologies FMOD Studio API
FSBank_Init

Initialize the FSBank system.

C Syntax

FSBANK_RESULT FSBank_Init(
    FSBANK_FSBVERSION version,
    FSBANK_INITFLAGS flags,
    unsigned int numSimultaneousJobs,
    const char *cacheDirectory
);

**Parameters**

*version*

FSB version, currently only `FSBANK_FSBVERSION_FSB5` is supported.

*flags*

Initialization flags which control how the system behaves.

*numSimultaneousJobs*

The maximum number of threads to create for parallel encoding. Set this to your number of CPU 'cores' for best performance.

*cacheDirectory*

Optional location to store the temporary cache files, default is a directory off the current working directory.
Return Values

If the function succeeds then the return value is **FSBANK_OK**. If the function fails then the return value will be one of the values defined in the **FSBANK_RESULT** enumeration.
See Also

- **FSBANK_FSBVERSION**
- **FSBANK_INITFLAGS**
- **FSBank_Release**
Firelight Technologies FMOD Studio API
FSBank_MemoryGetStats

Query the current and maximum memory usage of the FSBank system.

C Syntax

FSBANK_RESULT FSBank_MemoryGetStats(
    unsigned int *currentAllocated,
    unsigned int *maximumAllocated
);
Parameters

$currentAllocated$

Address of a variable that receives the currently allocated memory at time of call. Optional. Specify 0 or NULL to ignore.

$maximumAllocated$

Address of a variable that receives the maximum allocated memory since $FSBank_Init$. Optional. Specify 0 or NULL to ignore.
Return Values

If the function succeeds then the return value is FSBANK_OK.
If the function fails then the return value will be one of the values defined in the FSBANK_RESULT enumeration.
Firelight Technologies FMOD Studio API
**FSBank_MemoryInit**

Specifies a method for FSBank to allocate memory through callbacks.

**C Syntax**

```c
FSBANK_RESULT FSBank_MemoryInit(
    FSBANK_MEMORY_ALLOC_CALLBACK userAlloc,
    FSBANK_MEMORY_REALLOC_CALLBACK userRealloc,
    FSBANK_MEMORY_FREE_CALLBACK userFree
);
```
**Parameters**

*userAlloc*

Overrides the internal calls to alloc. Compatible with ANSI malloc().

*userRealloc*

Overrides the internal calls to realloc. Compatible with ANSI realloc().

*userFree*

Overrides the internal calls to free. Compatible with ANSI free().
Return Values

If the function succeeds then the return value is `FSBANK_OK`. If the function fails then the return value will be one of the values defined in the `FSBANK_RESULT` enumeration.
See Also

- FSBANK_MEMORY_ALLOC_CALLBACK
- FSBANK_MEMORY_REALLOC_CALLBACK
- FSBANK_MEMORY_FREE_CALLBACK
Firelight Technologies FMOD Studio API
FSBank_Release

Release the FSBank system, clean up used resources.

C Syntax

FSBANK_RESULT FSBank_Release();
Return Values

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

All progress items retrieved with `FSBank_FetchNextProgressItem` will be released by this function.
See Also

- FSBank_Init
- FSBank_FetchNextProgressItem
Firelight Technologies FMOD Studio API
FSBank_ReleaseProgressItem

Release memory associated with a progress item.

C Syntax

FSBANK_RESULT FSBank_ReleaseProgressItem(
    const FSBANK_PROGRESSITEM *progressItem
);
Parameters

`progressItem`

One status update about the progress of a particular subsound.
Return Values

If the function succeeds then the return value is **FSBANK_OK**. If the function fails then the return value will be one of the values defined in the **FSBANK_RESULT** enumeration.
See Also

- FSBank_FetchNextProgressItem
- FSBANK_PROGRESSITEM
Firelight Technologies FMOD Studio API
Callbacks

FSBANK_MEMORY_ALLOC_CALLBACK
FSBANK_MEMORY_FREE_CALLBACK
FSBANK_MEMORY_REALLOC_CALLBACK
Firelight Technologies FMOD Studio API
FSBANK_MEMORY_ALLOC_CALLBACK

Callback to allocate a block of memory.

C/C++ Syntax

```c
void * FSBANK_CALLBACK FSBANK_MEMORY_ALLOC_CALLBACK(
    unsigned int size,
    unsigned int 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

- **FSBank_MemoryInit**
- **FSBANK_MEMORY_REALLOC_CALLBACK**
- **FSBANK_MEMORY_FREE_CALLBACK**
Firelight Technologies FMOD Studio API
FSBANK_MEMORY_FREE_CALLBACK

Callback to free a block of memory.

C/C++ Syntax

```c
void FSBANK_CALLBACK FSBANK_MEMORY_FREE_CALLBACK(
    void *ptr,
    unsigned int 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

- `FSBank_MemoryInit`
- `FSBANK_MEMORY_ALLOC_CALLBACK`
- `FSBANK_MEMORY_REALLOC_CALLBACK`
Firelight Technologies FMOD Studio API
FSBANK_MEMORY_REALLOC_CALLBACK

Callback to re-allocate a block of memory to a different size.

C/C++ Syntax

```c
void * FSBANK_CALLBACK FSBANK_MEMORY_REALLOC_CALLBACK(
    void *ptr,
    unsigned int size,
    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.

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

- FSBank_MemoryInit
- FSBank_MEMORY_ALLOC_CALLBACK
- FSBank_MEMORY_FREE_CALLBACK
Firelight Technologies FMOD Studio API
Structures

FSBANK_PROGRESSITEM FSBANK_STATEDATA_FAILED
FSBANK_STATEDATA_WARNING
FSBANK_SUBSOUND
Firelight Technologies FMOD Studio API
FSBANK_PROGRESSITEM

Status information describing the progress of a build.

C/C++ Syntax

typedef struct {
    int subSoundIndex;
    int threadIndex;
    FSBANK_STATE state;
    const void *stateData;
} FSBANK_PROGRESSITEM;
Members

subSoundIndex

Index into the subsound list passed to FSBank_Build that this update relates to (-1 indicates no specific subsound).

threadIndex

Which thread index is serving this update (-1 indicates FSBank_Build / main thread).

state

Progress through the encoding process.

stateData

Cast to state specific data structure for extra information.
See Also

- **FSBank_Build**
- **FSBank_FetchNextProgressItem**
- **FSBank_ReleaseProgressItem**
- **FSBANK_STATE**
FSBANK_STATEDATA_FAILED

Extra state data for FSBANK_STATE_FAILED

C/C++ Syntax

typedef struct {
    FSBANK_RESULT errorCode;
    char errorString[256];
} FSBANK_STATEDATA_FAILED;
Members

errorCode
Error result code.

errorString
Description for error code.
Remarks

Cast stateData in FSBANK_PROGRESSITEM to this struct if the state is FSBANK_STATE_FAILED
See Also

- FSBANK_STATE_FAILED
- FSBANK_PROGRESSITEM
Firelight Technologies FMOD Studio API
FSBANK_STATEDATA_WARNING

Extra state data for FSBANK_STATEDATA_WARNING

C/C++ Syntax

typedef struct {
    FSBANK_RESULT warnCode;
    char warningString[256];
} FSBANK_STATEDATA_WARNING;
Members

warnCode

Warning result code.

warningString

Description for warning code.
Remarks

Cast stateData in FSBANK_PROGRESSITEM to this struct if the state is FSBANK_STATE_WARNING
See Also

- FSBANK_STATE_WARNING
- FSBANK_PROGRESSITEM
Firelight Technologies FMOD Studio API
FSBANK_SUBSOUND

Representation of how to encode a single subsound in the final FSB.

C/C++ Syntax

typedef struct {
    const char* const *fileNames;
    const void* const *fileData;
    const unsigned int *fileDataLengths;
    unsigned int numFiles;
    FSBANK_BUILDFLAGS overrideFlags;
    unsigned int overrideQuality;
    float desiredSampleRate;
    float percentOptimizedRate;
} FSBANK_SUBSOUND;
Members

fileNames

List of file names (instead of FSBANK_SUBSOUND::fileData) used to produce an interleaved sound.

fileData

List of file data pointers (instead of FSBANK_SUBSOUND::fileNames) used to produce an interleaved sound.

fileDataLengths

List of file data lengths corresponding to the items in the FSBANK_SUBSOUND::fileData list.

numFiles

Number of items in either FSBANK_SUBSOUND::fileData / FSBANK_SUBSOUND::fileDataLengths or FSBANK_SUBSOUND::fileNames.

overrideFlags

Flags that will reverse the equivalent flags passed to FSBank_Build.

overrideQuality

Override the quality setting passed to FSBank_Build.

desiredSampleRate

Resample to this sample rate (ignores optimize sample rate setting), up to 192000Hz.

percentOptimizedRate

If using FSBANK_BUILD_OPTIMIZESAMPLERATE, this is the percentage of
that rate to be used, up to 100.0%.
See Also

- **FSBank_Build**
- **FSBANK_BUILD_OPTIMIZESAMPLERATE**
- **FSBANK_SPEAKERMAP**
- **FSBANK_BUILDFLAGS**
Firelight Technologies FMOD Studio API
Defines

FSBANK_BUILDFLAGS  FSBANK_INITFLAGS
Firelight Technologies FMOD Studio API
FSBANK_BUILDFLAGS

Bit fields to use with `FSBank_Build` and in `FSBANK_SUBSOUND` to control how subsounds are encoded.

C/C++ Syntax

```c
#define FSBANK_BUILD_DEFAULT 0x00000000
#define FSBANK_BUILD_DISABLESYNCPOINTS 0x00000001
#define FSBANK_BUILD_DONTLOOP 0x00000002
#define FSBANK_BUILD_FILTERHIGHFREQ 0x00000004
#define FSBANK_BUILD_DISABLESEEKING 0x00000008
#define FSBANK_BUILD_OPTIMIZESAMPLERATE 0x00000010
#define FSBANK_BUILD_FSB5_DONTWRITENAMES 0x00000080
#define FSBANK_BUILD_NOGUID 0x00000100
#define FSBANK_BUILD_WRITEPEAKVOLUME 0x00000200
#define FSBANK_BUILD_OVERRIDE_MASK (FSBANK_BUILD_DISABLESYNCPOINTS | FSBANK_BUILD_DONTLOOP |)
#define FSBANK_BUILD_CACHE_VALIDATION_MASK (FSBANK_BUILD_DONTLOOP |)
```
Values

FSBANK_BUILD_DEFAULT

Build with default settings.

FSBANK_BUILD_DISABLESYNCPOINTS

Disable the storing of syncpoints in the output

FSBANK_BUILD_DONTLOOP

Disable perfect loop encoding and sound stretching. Removes chirps from the start of oneshot MP2, MP3 and IMAADPCM sounds.

FSBANK_BUILD_FILTERHIGHFREQ

XMA only. Enable high frequency filtering.

FSBANK_BUILD_DISABLESEEKING

XMA only. Disable seek tables to save memory.

FSBANK_BUILD_OPTIMIZESAMPLERATE

Attempt to optimize the sample rate down. Ignored if format is MP2, MP3 or if FSB4 basic headers flag is used.

FSBANK_BUILD_FSB5_DONTWRIITENAMES

FSB5 format only. Do not write out a names chunk to the FSB to reduce file size.

FSBANK_BUILD_NOGUID

FSB5 format only. Write out a null GUID for the FSB header. The runtime will not use header caching for these FSB files.

FSBANK_BUILD_WRITEPEAKVOLUME
FSB5 format only. Write peak volume for all subsounds.

**FSBANK_BUILD_OVERRIDE_MASK**

Build flag mask that specifies which settings can be overridden per subsound.

**FSBANK_BUILD_CACHE_VALIDATION_MASK**

Build flag mask that specifies which settings (when changed) invalidate a cache file.
See Also

- FSBank_Init
- FSBANK_SUBSOUND
Firelight Technologies FMOD Studio API
FSBANK_INITFLAGS

Bit fields to use with FSBank_Init to control the general operation of the library.

C/C++ Syntax

#define FSBANK_INIT_NORMAL 0x00000000
#define FSBANK_INIT_IGNOREERRORS 0x00000001
#define FSBANK_INIT_WARNINGSASERRORS 0x00000002
#define FSBANK_INIT_CREATEINCLUDEHEADER 0x00000004
#define FSBANK_INIT_DONTLOADCACHEFILES 0x00000008
#define FSBANK_INIT_GENERATEPROGRESSITEMS 0x00000010
Values

FSBANK_INIT_NORMAL

Initialize normally.

FSBANK_INIT_IGNOREERRORS

Ignore individual subsound build errors, continue building for as long as possible.

FSBANK_INIT_WARNINGSASERRORS

Treat any warnings issued as errors.

FSBANK_INIT_CREATEINCLUDEHEADER

Create C header files with #defines defining indices for each member of the FSB.

FSBANK_INIT_DONTLOADCACHEFILES

Ignore existing cache files.

FSBANK_INIT_GENERATEPROGRESSITEMS

Generate status items that can be queried by another thread to monitor the build progress and give detailed error messages.
See Also

- FSBank_Init
Firelight Technologies FMOD Studio API
Enumerations

FSBANK_FORMAT  FSBANK_FSBVERSION  
FSBANK_RESULT  
FSBANK_STATE  

Firelight Technologies FMOD Studio API
FSBANK_FORMAT

Compression formats available for encoding

C/C++ Syntax

typedef enum {
    FSBANK_FORMAT_PCM,
    FSBANK_FORMAT_PCM_BIGENDIAN,
    FSBANK_FORMAT_XMA,
    FSBANK_FORMAT_AT9_PSVITA,
    FSBANK_FORMAT_AT9_PS4,
    FSBANK_FORMAT_VORBIS,
    FSBANK_FORMAT_FADPCM,
    FSBANK_FORMAT_MAX
} FSBANK_FORMAT;
Values

FSBANK_FORMAT_PCM
PCM (1:1) All platforms.

FSBANK_FORMAT_PCM_BIGENDIAN
PCM Big Endian (1:1) Xbox360 and PS3 only.

FSBANK_FORMAT_XMA
XMA (VBR) Xbox360 / XboxOne only (hardware). Depends on xmaencoder.

FSBANK_FORMAT_AT9_PSVITA
ATRAC9 (CBR) PSVita only (hardware). Depends on libatrac9.

FSBANK_FORMAT_AT9_PS4
ATRAC9 (CBR) PS4 only (hardware). Depends on libatrac9.

FSBANK_FORMAT_VORBIS
Vorbis (VBR) All platforms. Depends on libvorbis.

FSBANK_FORMAT_FADPCM
FMOD ADPCM (3.5:1) All platforms.

FSBANK_FORMAT_MAX
Upper bound for this enumeration, for use with validation.
See Also

- FSBank_Build
Firelight Technologies FMOD Studio API
FSBANK_FSBVERSION

Version of FSB to write out.

C/C++ Syntax

typedef enum {
    FSBANK_FSBVERSION_FSB5,
    FSBANK_FSBVERSION_MAX
} FSBANK_FSBVERSION;
Values

*FSBANK_FSBVERSION_FSB5*

Produce FSB version 5 files.

*FSBANK_FSBVERSION_MAX*

Upper bound for this enumeration, for use with validation.
See Also

- FSBank_Init
Firelight Technologies FMOD Studio API
Error codes returned from every function.

C/C++ Syntax

typedef enum {
    FSBANK_OK,
    FSBANK_ERR_CACHE_CHUNKNOTFOUND,
    FSBANK_ERR_CANCELLED,
    FSBANK_ERR_CANNOT_CONTINUE,
    FSBANK_ERR_ENCODER,
    FSBANK_ERR_ENCODER_INIT,
    FSBANK_ERR_ENCODER_NOTSUPPORTED,
    FSBANK_ERR_FILE_OS,
    FSBANK_ERR_FILE_NOTFOUND,
    FSBANK_ERR_FMOD,
    FSBANK_ERR_INITIALIZED,
    FSBANK_ERR_INVALID_FORMAT,
    FSBANK_ERR_INVALID_PARAM,
    FSBANK_ERR_MEMORY,
    FSBANK_ERR_UNINITIALIZED,
    FSBANK_ERR_WRITER_FORMAT,
    FSBANK_WARN_CANNOTLOOP,
    FSBANK_WARN_IGNORED_FILTERHIGHFREQ,
    FSBANK_WARN_IGNORED_DISABLESEEKING,
    FSBANK_WARN_FORCED_DONTWRITENAMES,
    FSBANK_ERR_ENCODER_FILE_NOTFOUND,
    FSBANK_ERR_ENCODER_FILE_BAD
} FSBANK_RESULT;
Values

FSBANK_OK
No errors.

FSBANK_ERR_CACHE_CHUNKNOTFOUND
An expected chunk is missing from the cache, perhaps try deleting cache files.

FSBANK_ERR_CANCELLED
The build process was cancelled during compilation by the user.

FSBANK_ERR_CANNOT_CONTINUE
The build process cannot continue due to previously ignored errors.

FSBANK_ERR_ENCODER
Encoder for chosen format has encountered an unexpected error.

FSBANK_ERR_ENCODER_INIT
Encoder initialization failed.

FSBANK_ERR_ENCODER_NOTSUPPORTED
Encoder for chosen format is not supported on this platform.

FSBANK_ERR_FILE_OS
An operating system based file error was encountered.

FSBANK_ERR_FILE_NOTFOUND
A specified file could not be found.

FSBANK_ERR_FMODO
Internal error from FMOD sub-system.

FSBANK_ERR_INITIALIZED

Already initialized.

FSBANK_ERR_INVALID_FORMAT

The format of the source file is invalid.

FSBANK_ERR_INVALID_PARAM

An invalid parameter has been passed to this function.

FSBANK_ERR_MEMORY

Ran out of memory.

FSBANK_ERR_UNINITIALIZED

Not initialized yet.

FSBANK_ERR_WRITER_FORMAT

Chosen encode format is not supported by this FSB version.

FSBANK_WARN_CANNOTLOOP

Source file is too short for seamless looping. Looping disabled.

FSBANK_WARN_IGNORED_FILTERHIGHFREQ

FSBANK_BUILD_FILTERHIGHFREQ flag ignored: feature only supported by XMA format.

FSBANK_WARN_IGNORED_DISABLESEEKING

FSBANK_BUILD_DISABLESEEKING flag ignored: feature only supported by XMA format.

FSBANK_WARN_FORCED_DONTWRITENAMES
**FSBANK_BUILD_FSB5_DONTWRITENAMES** flag forced: cannot write names when source is from memory.

**FSBANK_ERR_ENCODER_FILE_NOTFOUND**

External encoder dynamic library not found

**FSBANK_ERR_ENCODER_FILE_BAD**

External encoder dynamic library could not be loaded, possibly incorrect binary format, incorrect architecture, file corruption
Firelight Technologies FMOD Studio API
FSBANK_STATE

Current state during the build process.

C/C++ Syntax

typedef enum {
    FSBANK_STATE_DECODING,
    FSBANK_STATE_ANALYSING,
    FSBANK_STATE_PREPROCESSING,
    FSBANK_STATE_ENCODING,
    FSBANK_STATE_WRITING,
    FSBANK_STATE_FINISHED,
    FSBANK_STATE_FAILED,
    FSBANK_STATE_WARNING
} FSBANK_STATE;
**Values**

*FSBANK_STATE_DECODING*

Decode a file to usable raw sample data.

*FSBANK_STATE_ANALYSING*

Scan sound data for details (such as optimized sample rate).

*FSBANK_STATE_PREPROCESSING*

Prepares sound data for encoder.

*FSBANK_STATE_ENCODING*

Pass the sample data to the chosen encoder.

*FSBANK_STATE_WRITING*

Write encoded data into an FSB.

*FSBANK_STATE_FINISHED*

Process complete.

*FSBANK_STATE_FAILED*

An error has occurred, check data (as FSBANK_STATE_DATA_FAILED) for details.

*FSBANK_STATE_WARNING*

A warning has been issued, check data (as FSBANK_STATE_DATA_WARN) for details.
See Also

- FSBANK_PROGRESSITEM
Firelight Technologies FMOD Studio API
Performance Reference

- Android
- IOS
- Mac
- Linux
- Windows Store Apps
- Windows Phone
- Windows
- Universal Windows Platform
Firelight Technologies FMOD Studio API
Performance Reference

This document is a companion for the performance tutorial document and serves as a quick reference of facts targeting this platform.
Format Choice

Each compression format provided in FMOD has a reason for being included, the below list will detail our recommendations for this platform. Formats listed as primary are considering the best choice, secondary formats should only be considered if the primary doesn't satisfy your requirements.

- **FADPCM**: Primary format for all sounds.
- **Vorbis**: Secondary format for long streams if FADPCM compression is too low.
- **PCM**: Secondary format for short sounds if FADPCM cost is too high.
- **XMA**: Unavailable.
- **AT9**: Unavailable.
Voice Count

To give developers an idea about the costs of a particular format we provide synthetic benchmark results. These results are based on simple usage of the FMOD Studio API using recommended configuration settings.

Due to the CPU governor that controls the power saving features of the device, getting accurate CPU numbers requires rooting the device and setting the CPU frequency to maximum.

Settings

- **Voice count:** 32
- **Sample rate:** 24KHz
- **Speaker mode:** Stereo
- **DSP block size:** 512 samples

Test Device: A

- **CPU:** Krait 400 @ 2.26 GHz (Nexus 5)
- **OS:** 4.4.4

Results: A

- **DSP with Vorbis:** 19.2% (+/- 2.0%)
- **DSP with FADPCM:** 3.2% (+/- 0.4%)
- **DSP with PCM:** 1.0% (+/- 0.2%)
- **Update at 60 FPS:** 1.3% (+/- 0.1%)

Test Device: B

- **CPU:** ARM Cortex-A8 @ 1 GHz (Nexus S)
- **OS:** 4.1.1

Results: B

- **DSP with Vorbis:** N/A
- **DSP with FADPCM:** 17.1% (+/- 2.7%)
- **DSP with PCM:** 12.4% (+/- 1.5%)
- **Update at 60 FPS:** 2.6% (+/- 2.9%)
Firelight Technologies FMOD Studio API
Performance Reference

This document is a companion for the performance tutorial document and serves as a quick reference of facts targeting this platform.
Format Choice

Each compression format provided in FMOD has a reason for being included, the below list will detail our recommendations for this platform. Formats listed as primary are considering the best choice, secondary formats should only be considered if the primary doesn't satisfy your requirements.

- **FADPCM**: Primary format for all sounds.
- **Vorbis**: Secondary format for long streams if FADPCM compression is too low.
- **PCM**: Secondary format for short sounds if FADPCM cost is too high.
- **AAC**: Special format for long streams, single hardware assisted codec available for .MP4 / .M4A files.
- **XMA**: Unavailable.
- **AT9**: Unavailable.
Voice Count

To give developers an idea about the costs of a particular format we provide synthetic benchmark results. These results are based on simple usage of the FMOD Studio API using recommended configuration settings.

Settings

- **Voice count:** 32
- **Sample rate:** 24KHz
- **Speaker mode:** Stereo
- **DSP block size:** 1024 samples

Test Device: A

- **CPU:** Apple A7 @ 1.3 GHz (iPhone 5S)
- **OS:** 8.0

Results: A

- **DSP with Vorbis:** 13.7% (+/- 1.3%)
- **DSP with FADPCM:** 3.3% (+/- 0.3%)
- **DSP with PCM:** 1.1% (+/- 0.2%)
- **Update at 60 FPS:** 0.9% (+/- 0.1%)

Test Device: B

- **CPU:** ARM Cortex-A9 @ 0.8 GHz (iPhone 4S)
- **OS:** 7.1.2

Results: B

- **DSP with Vorbis:** N/A
- **DSP with FADPCM:** 11.9% (+/- 0.8%)
- **DSP with PCM:** 6.9% (+/- 0.6%)
- **Update at 60 FPS:** 2.8% (+/- 0.5%)
Firelight Technologies FMOD Studio API
Performance Reference

This document is a companion for the performance tutorial document and serves as a quick reference of facts targeting this platform.
Format Choice

Each compression format provided in FMOD has a reason for being included, the below list will detail our recommendations for this platform. Formats listed as primary are considering the best choice, secondary formats should only be considered if the primary doesn't satisfy your requirements.

- **Vorbis**: Primary format for all sounds.
- **FADPCM**: Secondary format if Vorbis CPU usage is too high for low spec machines.
- **PCM**: Not recommended.
- **XMA**: Unavailable.
- **AT9**: Unavailable.
Voice Count

To give developers an idea about the costs of a particular format we provide synthetic benchmark results. These results are based on simple usage of the FMOD Studio API using recommended configuration settings.

Settings

- **Voice count:** 64
- **Sample rate:** 48KHz
- **Speaker mode:** Stereo
- **DSP block size:** 512 samples

Test Device

- **CPU:** Intel Core i7 3615QM @ 2.3 GHz (Late 2012 Mac Mini)
- **OS:** 10.8.5

Results

- **DSP with Vorbis:** 31.0% (+/- 3.6%)
- **DSP with FADPCM:** 8.7% (+/- 0.8%)
- **DSP with PCM:** 5.4% (+/- 0.7%)
- **Update at 60 FPS:** 2.0% (+/- 0.1%)
Firelight Technologies FMOD Studio API
Performance Reference

This document is a companion for the performance tutorial document and serves as a quick reference of facts targeting this platform.
Format Choice

Each compression format provided in FMOD has a reason for being included, the below list will detail our recommendations for this platform. Formats listed as primary are considering the best choice, secondary formats should only be considered if the primary doesn't satisfy your requirements.

- **Vorbis**: Primary format for all sounds.
- **FADPCM**: Secondary format if Vorbis CPU usage is too high for low spec machines.
- **PCM**: Not recommended.
- **XMA**: Unavailable.
- **AT9**: Unavailable.
Voice Count

To give developers an idea about the costs of a particular format we provide synthetic benchmark results. These results are based on simple usage of the FMOD Studio API using recommended configuration settings.

Settings

- **Voice count:** 64
- **Sample rate:** 48KHz
- **Speaker mode:** Stereo
- **DSP block size:** 1024 samples

Test Device

- **CPU:** Intel Core i7 920 @ 2.67 GHz (Virtual Box)
- **OS:** Ubuntu 13.10

Results

- **DSP with Vorbis:** 26.8% (± 5.6%)
- **DSP with FADPCM:** 7.2% (± 1.3%)
- **DSP with PCM:** 3.3% (± 0.9%)
- **Update at 60 FPS:** 0.8% (± 0.2%)
Firelight Technologies FMOD Studio API
Performance Reference

This document is a companion for the performance tutorial document and serves as a quick reference of facts targeting this platform.
Format Choice

Each compression format provided in FMOD has a reason for being included, the below list will detail our recommendations for this platform. Formats listed as primary are considering the best choice, secondary formats should only be considered if the primary doesn't satisfy your requirements.

- **Vorbis**: Primary format for all sounds.
- **FADPCM**: Secondary format if Vorbis CPU usage is too high for low spec machines.
- **PCM**: Not recommended.
- **XMA**: Unavailable.
- **AT9**: Unavailable.
Voice Count

To give developers an idea about the costs of a particular format we provide synthetic benchmark results. These results are based on simple usage of the FMOD Studio API using recommended configuration settings.

Settings

- **Voice count:** 64
- **Sample rate:** 48KHz
- **Speaker mode:** Stereo
- **DSP block size:** 1024 samples

Test Device

- **CPU:** Intel Core 2 Duo E6700 @ 2.66 GHz
- **OS:** Windows 8.1

Results

- **DSP with Vorbis:** 25.6% (+/- 2.0%)
- **DSP with FADPCM:** 5.2% (+/- 0.3%)
- **DSP with PCM:** 3.0% (+/- 0.2%)
- **Update at 60 FPS:** 0.8% (+/- 0.2%)
Firelight Technologies FMOD Studio API
Performance Reference

This document is a companion for the performance tutorial document and serves as a quick reference of facts targeting this platform.
Format Choice

Each compression format provided in FMOD has a reason for being included, the below list will detail our recommendations for this platform. Formats listed as primary are considering the best choice, secondary formats should only be considered if the primary doesn't satisfy your requirements.

- **FADPCM**: Primary format for all sounds.
- **Vorbis**: Secondary format for long streams if FADPCM compression is too low.
- **PCM**: Secondary format for short sounds if FADPCM cost is too high.
- **XMA**: Unavailable.
- **AT9**: Unavailable.
Voice Count

To give developers an idea about the costs of a particular format we provide synthetic benchmark results. These results are based on simple usage of the FMOD Studio API using recommended configuration settings.

Settings

- **Voice count**: 32
- **Sample rate**: 24KHz
- **Speaker mode**: Stereo
- **DSP block size**: 1024 samples

Test Device

- **CPU**: Snapdragon 400 @ 1.2 GHz (Lumia 630)
- **OS**: 8.1

Results

- **DSP with Vorbis**: N/A
- **DSP with FADPCM**: 14.0% (+/- 1.8%)
- **DSP with PCM**: 5.7% (+/- 0.6%)
- **Update at 60 FPS**: 3.1% (+/- 0.6%)
Firelight Technologies FMOD Studio API
Performance Reference

This document is a companion for the performance tutorial document and serves as a quick reference of facts targeting this platform.
Format Choice

Each compression format provided in FMOD has a reason for being included, the below list will detail our recommendations for this platform. Formats listed as primary are considering the best choice, secondary formats should only be considered if the primary doesn't satisfy your requirements.

- **Vorbis**: Primary format for all sounds.
- **FADPCM**: Secondary format if Vorbis CPU usage is too high for low spec machines.
- **PCM**: Not recommended.
- **XMA**: Unavailable.
- **AT9**: Unavailable.
Voice Count

To give developers an idea about the costs of a particular format we provide synthetic benchmark results. These results are based on simple usage of the FMOD Studio API using recommended configuration settings.

Settings

- **Voice count**: 64
- **Sample rate**: 48KHz
- **Speaker mode**: Stereo
- **DSP block size**: 1024 samples

Test Device

- **CPU**: Intel Core 2 Duo E6700 @ 2.66 GHz
- **OS**: Windows 8.1

Results

- **DSP with Vorbis**: 25.6% (+/- 2.0%)
- **DSP with FADPCM**: 5.2% (+/- 0.3%)
- **DSP with PCM**: 3.0% (+/- 0.2%)
- **Update at 60 FPS**: 0.8% (+/- 0.2%)