Scream Library Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

About This Document

This document is a detailed guide to using the Scream library, as applicable to the NGS synthesizer running on the PlayStation®Vita platform and the NGS2 synthesizer running on the PlayStation®4 platform.

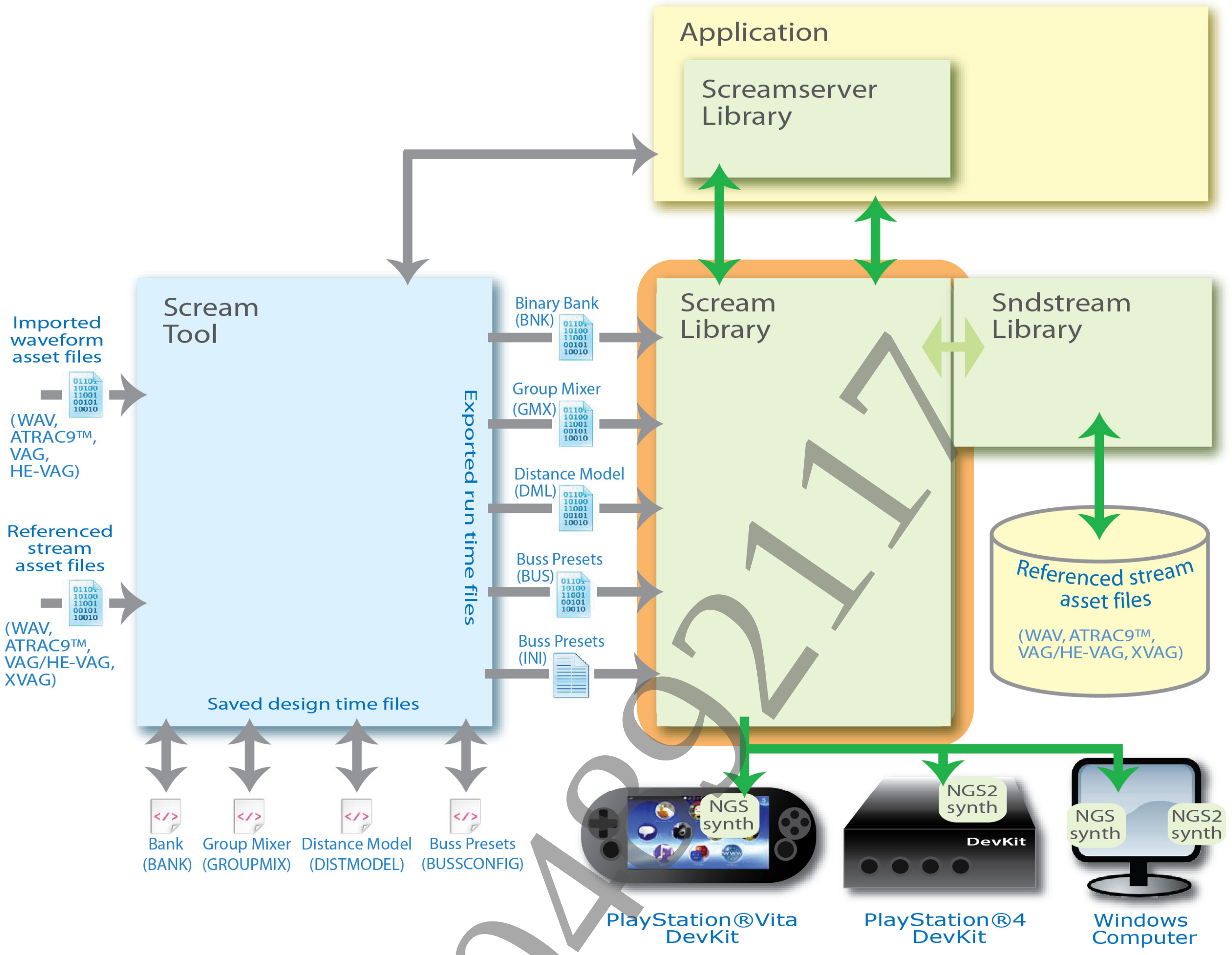
1 Introduction

Scream is a runtime library for delivering interactive audio content for the PlayStation®Vita and PlayStation®4 platforms. Scream consists of the following components:

- **Scream Runtime Library** – Controls the underlying audio synthesizer, and coordinates playback of Sounds contained in Banks (the basic data format used by Scream). Documented in this manual (the *Scream Library Overview*) and in the *Scream Library Reference* documents.
- **Sndstream Runtime Library** – An add-on runtime library for streaming audio files from external media. Sndstream provides a range of features for manipulating, queuing, and transitioning streams. Once initialized, Streams can also be manipulated by the Scream runtime (along with Scream Sounds). Documented in the *Sndstream Library Overview* and the *Sndstream Library Reference*.
- **Scream Tool** – An environment used by audio designers to create Scream Sounds. A Sound consists of raw waveform data plus a script that defines how to play it at run time. Sounds are exported from Scream Tool in Banks, and then loaded into the Scream runtime for playback. Scream Tool also allows audio designers to setup and control streaming audio files to be played back by Sndstream. Documented in the *Scream Tool Getting Started Guide* and the *Scream Tool Help*.
- **Scream Tool Auditioning Servers** – Connected to Scream Tool, the servers simulate the audio capabilities of the target platforms, allowing audio designers to audition game audio content.
- **Screamserver Runtime Library** – By including Screamserver in a special audio test build, audio designers can connect Scream Tool to a running instance of a game. This is useful for interactively adjusting Sound parameters and fine-tuning the mix of audio elements during the latter stages of the development cycle. Documented in the *Screamserver Library Reference*.
- **Scream BankMerge/Build Utility** – A command-line tool for merging and exporting Bank, effect preset, and distance model files as part of a build process. Documented in the *Scream BankMerge/Build Utility User's Guide*.

Figure 1 depicts topology of the Scream audio system, with Scream outlined in orange.

Figure 1 Scream Audio System Topology



Summary of Features

Features of the Scream runtime library for PlayStation®Vita and PlayStation®4 include:

- processing of Sound scripts created in Scream Tool
- processing of CCSounds, controlling scripted Sound parameters and playback based on game-manipulated variables
- grouping of Sounds; manipulation – of volume, pause, stop, voice allocation, speaker target exclusion – per Group
- I3DL2 reverb, delay (NGS2 only), and chorus (NGS2 only) auxiliary buss effects
- distance modeling per Group, Sound, or asset Grain
- built-in Doppler effect
- automated parameter change functions; allow smooth pan, gain, pitch transposition, and pitchbend changes to Sounds over a specified time
- individual, collective, and atomic Sound local register setting and retrieval functions
- volume ducking
- variable-speed/pitch audio replay
- sound spatialization and movement utility functions
- pre-master submix busses effects

- master buss effects
- priority-based voice management

Basic Workflow

Audio designers import audio assets and create scripts in Scream Tool that specify how to manipulate a Sound. A Sound script can be simple, such as a single one-shot sound effect, or complex, such as a script that starts child Sounds based on feedback from the game engine. Audio designers export Sounds (consisting of audio assets and associated scripts) in the form of platform-specific binary Bank files (BNK), which are loaded into the Scream runtime component. In most cases, Scream's scripts and Bank format allow audio designers to provide Banks to audio programmers without the need for lengthy details about how and when Sounds should be played. This enables audio designers and audio programmers to focus on their respective tasks.

Platform Compatibility

This document and the Scream runtime library for the NGS and NGS2 synthesizers apply to game development on the PlayStation®Vita and PlayStation®4 platforms respectively.

Embedding into a Program

To embed the Scream and Screamserver libraries into a program, you must meet the following dependencies.

Scream Header and Library Files

Header and library files required for using Scream on the PlayStation®Vita and PlayStation®4 platforms are listed in Table 1.

Table 1 Required Scream Header and Library Files

Scream File	Description
`scream_top.h`	Prototypes for the general public interface, and definitions of general data structures. **Note:** Includes all Scream, Sndstream, and Screamserver header files.
`libSceScream.a`	The main Scream library.

Screamserver Library File

By including Screamserver in a special audio test build, you can enable audio designers to connect Scream Tool to a running instance of a game, allowing in-game editing of audio parameters. The header file required for using Screamserver on the PlayStation®Vita and PlayStation®4 platforms is included within `scream_top.h`. The library file is listed in Table 2.

Table 2 Required Screamserver Library File

Scream Server File	Description
`libSceScreamserver.a`	The main Screamserver library.

Load the NGS and NGS2 Modules

On the PlayStation®Vita platform, applications must first load the NGS module before initializing the Scream Runtime. On the PlayStation®4 platform, applications must first load the NGS2 module before initializing the Scream Runtime.

You use the `sceSysmoduleLoadModule()` function (in the NGS and NGS2 libraries) to load the NGS and NGS2 modules, specifying the appropriate constant as the function's only argument:

- NGS module: `SCE_SYSMODULE_NGS`
- NGS2 module: `SCE_SYSMODULE_NGS2`

Link with the NGS and NGS2 Libraries

Before initializing Scream you must link to the following libraries in build script(s):

Table 3 NGS and NGS2 Library Files

NGS	NGS2
- `libSceNgs` : NGS library	- `libSceNgs2` : NGS2 library
- `libSceAudio` : PlayStation®Vita audio library	- `libSceAudioOut` : PlayStation®4 audio library
- `libSceScream.a` : Scream NGS library	- `libSceScream.a` : Scream NGS2 library

Sample Programs

The PlayStation®Vita and PlayStation®4 SDKs include a number of Scream sample programs. With respect to platform SDK, the sample programs install to the following directories:

PlayStation®Vita	`%SCE_PSP2_SDK_DIR%/target/samples/sample_code/`
PlayStation®4	`%SCE_ORBIS_SDK_DIR%/target/samples/sample_code/`

The sample programs share common platform, audio initialization, and shutdown code, contained in the files:

- `sample_code/audio_video/api_libscream/common/audio_examples.h`
- `sample_code/audio_video/api_libscream/common/audio_examples.cpp`

The sample programs are usable as code sample resources. Source files are located in each sample's directory, and named *sample*.cpp (where *sample* is the name of each sample program).

Visual Studio Solutions Files

The `api_libscream` directory and the sample-specific subdirectories contain Visual Studio (VS) Solutions files for VS Pro. The main Solutions file launches a VS Solution that contains all the sample programs, and is named as follows:

PlayStation®Vita	`api_libscream.sln`
PlayStation®4	`api_libscream.sln`

Azimuth

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_azimuth`
PlayStation®4	`sample_code/audio_video/api_libscream/azimuth`

Scream Azimuth demonstrates panning a Sound. The sample plays a Sound with different azimuth values by setting and resetting the `SceScreamSoundParams` structure's *`azimuth`* member. The sample also demonstrates routing to custom player-specific outputs on the PlayStation®4 platform.

CCSound

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_ccsound`
PlayStation®4	`sample_code/audio_video/api_libscream/ccsound`

Scream CCSound demonstrates a CCSound used for a game vehicle. The sample sets initial values for the CCSound's input variables, plays the CCSound, and then resets input variables to manipulate the vehicle's engine speed, turn squeals, and an automatic weapon fired from the vehicle.

Distance Model

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_distance_model`

PlayStation®4	`sample_code/audio_video/api_libscream/distance_model`

Distance Model demonstrates loading a distance model file, and setting Sound and listener 3D positional coordinates to supply a Sound with distance input.

Doppler

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_doppler`
PlayStation®4	`sample_code/audio_video/api_libscream/doppler`

Scream Doppler demonstrates the Doppler effect, simulating the movement of a sound-producing object. The sample calls the `sceScreamGetDopplerPitchTranspose()` function to calculate pitch transposition amounts, and applies the output to the `SceScreamSoundParams` *`pitchTranspose`* member.

Filter

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_filter`
PlayStation®4	`sample_code/audio_video/api_libscream/filter`

Scream Filter demonstrates application of a filter to a Sound. The sample loads a Bank, and initializes a `SceScreamSoundParams` structure (including filter settings). It then plays a *Cannon* Sound – first without the assigned filter, and again with the assigned filter.

Groupmix

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_groupmix`
PlayStation®4	`sample_code/audio_video/api_libscream/groupmix`

Scream Group Mix demonstrates working with mix snapshots. The sample loads a group mixer file, then activates and deactivates snapshots it contains.

Groups

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_groups`
PlayStation®4	`sample_code/audio_video/api_libscream/groups`

Scream Groups demonstrates Group pause and volume ducker functionality. The sample loads a Bank, and initializes a `SceScreamSoundParams` structure. It first plays a *Title Music* Sound for a few seconds, and then pauses the Music Group (of which the *Title Music* Sound is constituent). With the Music Group paused, the program plays a Sound from another Group (a *GunHandling* Sound assigned to the SFX Group), then waits a few seconds and continues the Music Group (resuming the *Title Music* Sound). Following this, the program sets up a volume ducker; assigning SFX as the source Group, and Music as the target Group. It then plays the *Title Music* Sound again. Only this time, when the *GunHandling* Sound enters, rather than pausing the Music Group, the Music Group is ducked instead; reducing the overall gain level of the *Title Music* Sound while the *GunHandling* Sound plays.

Registers

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_registers`
PlayStation®4	`sample_code/audio_video/api_libscream/registers`

Scream Registers demonstrates working with global registers. The sample plays a *Weather Controller* Sound, then manipulates the value of global register 1, to which the Sound responds.

Reverb

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_reverb`
PlayStation®4	`sample_code/audio_video/api_libscream/reverb`

Scream Reverb demonstrates the application of a reverb effect to a Sound. The sample loads a Bank, and initializes a `SceScreamSoundParams` structure (including auxiliary send gain and destination settings).

An I3DL2 reverb instance is created, assigned to an auxiliary buss, and assigned a preset. The program then plays a Sound – first dry (without reverb), then wet (reverb only), and finally wet and dry (original signal plus reverb).

Script Speed

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_script_speed`
PlayStation®4	`sample_code/audio_video/api_libscream/script_speed`

Scream Script Speed demonstrates variable speed replay functionality. The sample plays a *drum loop* Sound, first at normal speed, then with the execution of the Sound's script speeded up, and finally with speeded up execution combined with corresponding pitch transposition.

Submix

PlayStation®Vita	`sample_code/audio_video/api_libscream/scream_submix`
PlayStation®4	`sample_code/audio_video/api_libscream/submix`

Scream Submix demonstrates use of pre-master submixes. The sample plays a Sound that is initially assigned to the Master buss. It plays the Sound again, this time assigning it to a pre-master submix, upon which an EQ effect is set up. The sample then plays the Sound a third time, assigning it to another pre-master submix, upon which a compressor effect is set up.

Documentation

Documentation consists of Reference and Overview documents for the Scream and Sndstream libraries, and a Reference document only for the (very small) Screamserver library. There is also a glossary document containing a reference to audio terms and file extensions used in the Scream documentation. The following is a complete list of Scream library documents:

- *Scream Library Overview* (this document) – a guide to programming the Scream library.
- *Scream Library Reference* – a reference to the Scream library API, used in conjunction with the NGS synthesizer running on the PlayStation®Vita platform.
- *Scream Library Reference* – a reference to the Scream library API, used in conjunction with the NGS2 synthesizer running on the PlayStation®4 platform.
- *Sndstream Library Overview* – a guide to programming the Sndstream library.
- *Sndstream Library Reference* – a reference to the Sndstream library API.
- *Screamserver Library Reference* – a reference to the Screamserver library API.
- *Scream BankMerge/Build Utility User's Guide* – a guide to using the BankMerge/Build utility to merge, convert, and re-build Bank, effect preset, and distance model files to application specifications.
- *Scream Audio Glossary* – a reference to terms and file extensions used in the Scream Tool and Scream library documentation.

Tutorials

Tutorials are provided for both Scream Tool and Scream Runtime. The Scream Tool tutorial appears in a chapter of the *Scream Tool Help*, which is installed with Scream Tool. The Scream Runtime Tutorial appears as an appendix in this document.

2 System Overview

The Scream runtime component plays and manipulates Sounds contained in loaded Banks. Scream processes all control information – including instructions directly from the API, and from Sound scripts. Scream can also recall designer-created presets for auxiliary, pre-master submix, master speakers, and master headset buss effects. Scream can also activate designer-created mix snapshots.

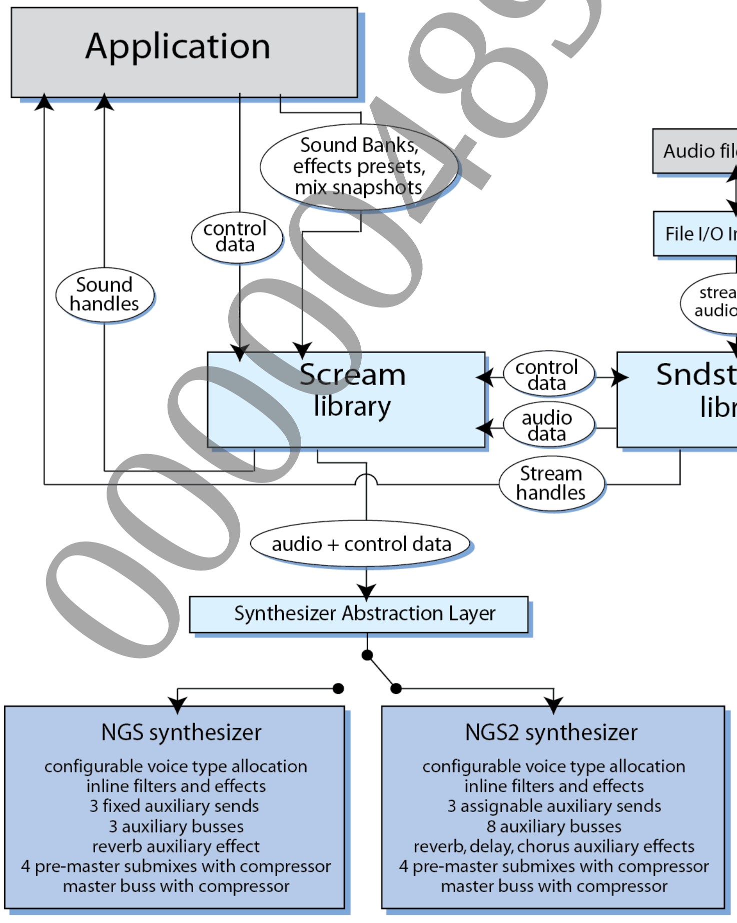
Scream supports the NGS rendering synthesizer on the PlayStation®Vita platform and the NGS2 rendering synthesizer on the PlayStation®4 platform.

On either platform you start Scream with a call to `sceScreamStartSoundSystemEx2()`. The common header file used by the sample programs (`audio_examples.h`) outlines initial configuration tasks for PlayStation®Vita and PlayStation®4 that you must complete before calling the `sceScreamStartSoundSystemEx2()` function. See Sample Programs for further details.

Audio and Control Data Flow

The Scream audio system comprises a number of components – runtime components, tool, synthesizers, and so on – each of which providing important functionality, as described in the "Introduction" chapter. Figure 2 depicts the run time flow of audio and control data between the various components. For a design time depiction of the components and data flow, see the "Auditioning" chapter in the *Scream Tool Help*.

Figure 2 Runtime Depiction of Scream Audio and Control Data Flow


Application
Sound Banks, effects presets, mix snapshots
control data
Sound handles
Audio files on disk
File I/O Interface
streamed audio data
data requests
Scream library
control data
audio data
Sndstream library
Stream handles
audio + control data
Synthesizer Abstraction Layer
NGS synthesizer
configurable voice type allocation
inline filters and effects
3 fixed auxiliary sends
3 auxiliary busses
reverb auxiliary effect
4 pre-master submixes with compressor
master buss with compressor
NGS2 synthesizer
configurable voice type allocation
inline filters and effects
3 assignable auxiliary sends
8 auxiliary busses
reverb, delay, chorus auxiliary effects
4 pre-master submixes with compressor
master buss with compressor

Scream Sounds and Synthesizer Voices

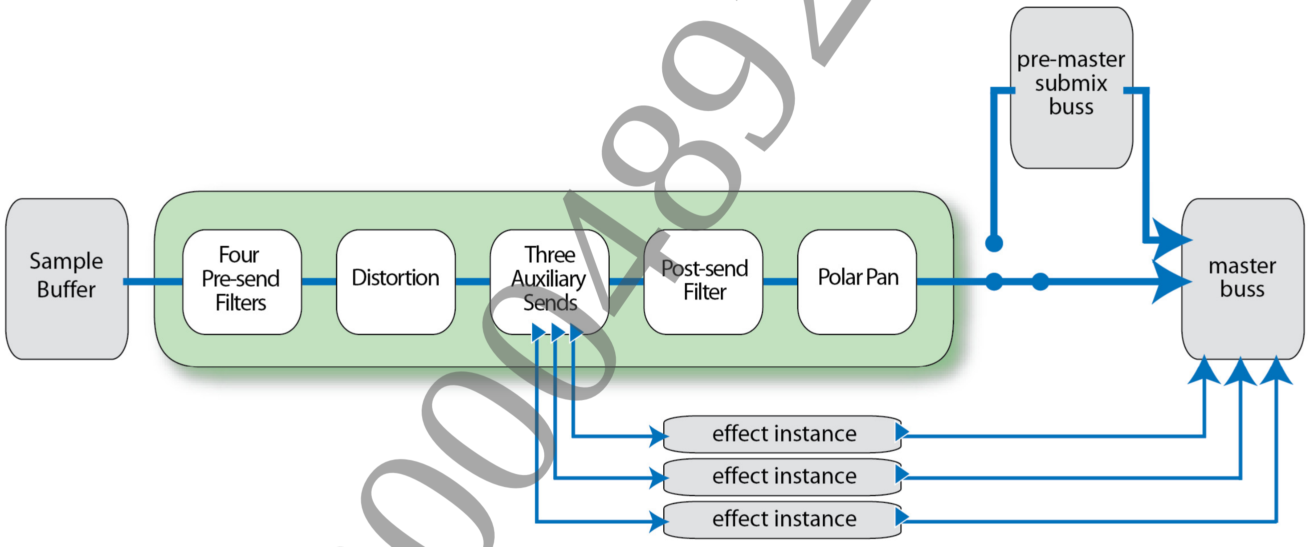
It is important to understand the one-to-many relationship between a Scream Sound and synthesizer voices. A *Waveform* encapsulates a single channel of audio content and maps to a single synth voice. A *Stream* can map to multiple synth voices, one for each audio channel. A Scream Sound can contain multiple *Waveforms* and *Streams*, and therefore may require many synth voices for playback.

When creating audio content in Scream Tool, audio designers set parameters for individual *Waveform* Grains – and thus set parameters on individual synth voices. However, from the Scream API you cannot address individual synth voices, only Sounds. When you set a parameter on a Scream Sound from the API, the setting is inherited by all voices used by the Sound. See Scream Tool and API Parameter Settings for further details.

Architecture of the NGS/NGS2 *Sampler Fat Mono* Voice

Scream supports the NGS/NGS2 input voice known as *Sampler Fat Mono*. The *Sampler Fat Mono* voice comprises modules that process – serially – incoming audio data, received from a sample buffer, both before and after it is optionally branched to three auxiliary send busses. Outgoing signal from a voice can be routed either to the master buss (the default) or to a pre-master submix buss. See Setting Sound Output Destinations, the "Working with Pre- and Post-Send Filters and Effects" chapter, and the "Working with Distance Models, Doppler, and Spatialization" chapter for further details. Signal flow through the NGS/NGS2 *Sampler Fat Mono* voice, and the modules it contains, are depicted in Figure 3.

Figure 3 NGS/NGS2 *Sampler Fat Mono* Voice: Signal Flow and Architecture



Note: On the PlayStation®4 platform, Sounds, reverb instances, and pre-master submix busses can also route signal to custom player-specific outputs. This signal flow is not depicted in Figure 3.

Voice Prioritization

Scream can play a maximum of 256 Sounds simultaneously. A Sound can sometimes require many synthesizer voices. With the total number of available voices on the NGS/NGS2 synthesizers being less than 256, a scenario can easily occur in which the number of required voices exceeds the number of available voices. In such a scenario, a prioritization mechanism manages the allocation of voices. Voice allocation has the following controls:

- **priority** – Audio designers set *Waveform* and *Stream* priority values in Scream Tool. This sets a priority for each allocated voice. Values range from 0 to 126; higher values indicate higher priorities. In a scenario where no free voices are available, voices with lower priority settings can be stolen by new voice allocation requests with higher priority settings. Note that a Sound can contain multiple *Waveforms* each with different priority settings – making some more susceptible to stealing than others. In such a case, it is quite possible that *Waveforms* with lower priorities will be stolen, while others, belonging to the same Sound, continue to play.
- **minRipoffTime** – The `SceScreamPlatformInitEx2` structure's *`minRipoffTime`* member specifies the minimum time (expressed in seconds) a voice must be allowed to play before it becomes eligible for stealing by new voice allocation requests. It defaults to (the System Defaults constant) `SCE_SCREAM_SND_DEFAULT_MIN_RIPOFF_TIME` (equal to 1.0).
- **...NO_STEAL / Keep Voice** – There are two ways to specify that voices cannot be stolen by subsequent voice allocation requests – from the Sndstream API, and in the Scream Tool *Waveform* and *Stream* properties. From the Sndstream API, you set the `SceScreamSndStartParams` structure's *`flags`* member to include the `SCE_SCREAM_SND_SS_START_VOICE_NO_STEAL` flag.

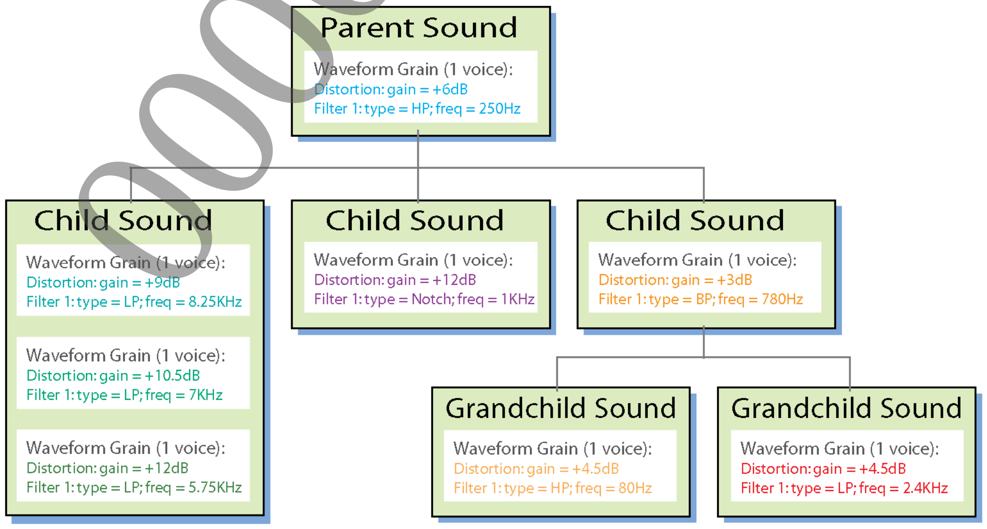
Note: Setting the ...`NO_STEAL` / `Keep Voice` option on a Sound or Stream has no bearing on its initial allocation of voice(s). Voice allocation is a function of the number of available voices and the priority of the requested voice(s).

- **scale with volume** – Voice allocation priorities can be further scaled according to volume/gain setting. Again, there are two ways to access this feature – from the Scream Tool *Waveform* and *Stream* properties, and from the Sndstream API. In Sndstream you set a value for the `SceScreamSndStartParams` *`priorityReductionScale`* member. Values range from 0.0 to 1.0. A value of 1.0 specifies maximum gain-based voice priority reduction; a value of 0.0 specifies zero gain-based voice priority reduction.

Parent and Child Sounds

In Scream Tool a Sound script can sprout one or more Child Sounds (using the *Start Child Sound* Grain). Further, Child Sounds can in turn sprout their own Child Sounds, forming a grandchild relationship with the original parent Sound, and so on. Each Parent or Child Sound usually requires one or more voices (some Sounds may be used for controlling other Sounds only, and do not contain *Waveform* or *Stream* Grains, and therefore do not use any voices). Figure 4 depicts an example of Parent/Child Sound relationships, with individual distortion and filter settings for each *Waveform* Grain (voice).

Figure 4 Parent/Child Sounds with Individual Distortion and Filter Settings



Parent Sound
Waveform Grain (1 voice):
Distortion: gain = +6dB
Filter 1: type = HP; freq = 250Hz
Child Sound
Waveform Grain (1 voice):
Distortion: gain = +9dB
Filter 1: type = LP; freq = 8.25KHz
Waveform Grain (1 voice):
Distortion: gain = +10.5dB
Filter 1: type = LP; freq = 7KHz
Waveform Grain (1 voice):
Distortion: gain = +12dB
Filter 1: type = LP; freq = 5.75KHz
Child Sound
Waveform Grain (1 voice):
Distortion: gain = +12dB
Filter 1: type = Notch; freq = 1KHz
Child Sound
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = BP; freq = 780Hz
Grandchild Sound
Waveform Grain (1 voice):
Distortion: gain = +4.5dB
Filter 1: type = HP; freq = 80Hz
Grandchild Sound
Waveform Grain (1 voice):
Distortion: gain = +4.5dB
Filter 1: type = LP; freq = 2.4KHz

Scream Tool and API Parameter Settings

As can be seen in Figure 4, a Sound may contain Child Sounds, and each Parent/Child Sound may contain one or more *Waveform* Grains – each Grain corresponds with an individual synthesizer voice, and potentially has its own unique parameter settings.

Parameter settings made from the API interact with parameter settings contained in the content (that is, saved with a Bank as exported from Scream Tool) in different ways, depending on whether the parameter is a member of the `SceScreamSynthParams` or `SceScreamSoundParams` structures. Regardless of this designation, an essential point to understand is that you cannot address individual voice parameters from the Scream API: parameters apply to a *Sound* — not a voice. This means that parameter settings made from the API are applied to *all* voices contained in the Sound's tree. API parameter settings (with respect to members of the `SceScreamSynthParams` structure) simply *override any and all* corresponding parameters in Grains contained in the Parent Sound or any of its descendants. API parameter settings with respect to members of the `SceScreamSoundParams` structure scale (in various ways) rather than replace *any and all* corresponding parameters in Grains contained in the Parent Sound. In some cases, this includes descendants too; in other cases, it is restricted to just the Parent Sound; see Setting Parameter Values for further details.

Consultation Point: Sound parameters settings made from the API can affect *all* corresponding parameters in Grains contained in the Parent Sound or its descendants. Consult with your audio designer before re-setting parameter values from the API.

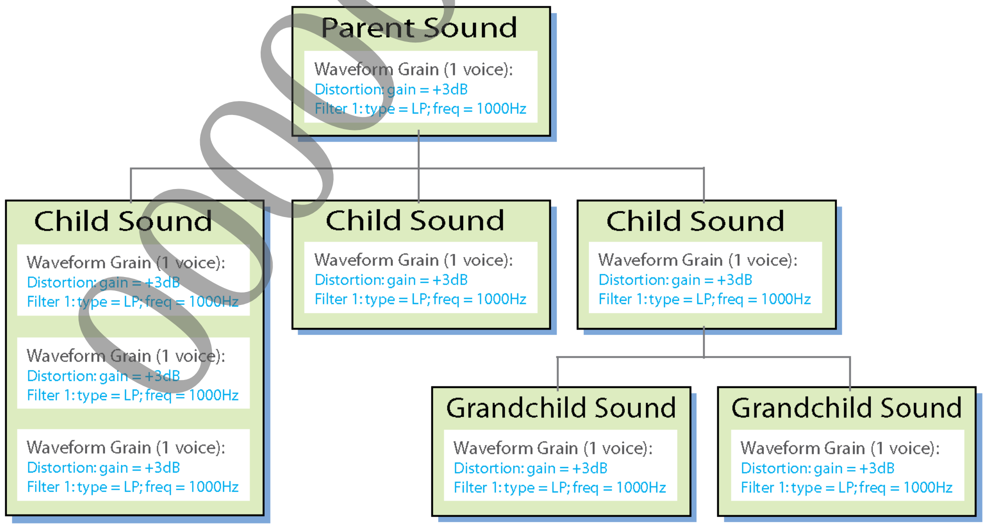
Setting SceScreamSynthParams Parameters

Setting any parameters that are members of the `SceScreamSynthParams` structure overrides any and all corresponding parameters in Grains contained in the Parent Sound or any of its descendants. For example, take the Sound depicted in Figure 4, and suppose you set distortion and filter parameters from the API as follows:

- Distortion: gain = 3 dB
- Filter 1: type = Low Pass; freq = 1000 Hz

All Grains contained in the Sound's tree would then inherit these settings, as shown in Figure 5.

Figure 5 Parent/Child Sounds with API-Overridden Distortion and Filter Settings


Parent Sound
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = LP; freq = 1000Hz
Child Sound
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = LP; freq = 1000Hz
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = LP; freq = 1000Hz
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = LP; freq = 1000Hz
Child Sound
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = LP; freq = 1000Hz
Child Sound
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = LP; freq = 1000Hz
Grandchild Sound
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = LP; freq = 1000Hz
Grandchild Sound
Waveform Grain (1 voice):
Distortion: gain = +3dB
Filter 1: type = LP; freq = 1000Hz

Setting SceScreamSoundParams Parameters

For details of how API settings with respect to `SceScreamSoundParams` members interact with corresponding settings made in the content, see Setting Parameter Values in the "Working with Sounds" chapter.

3 Configuration, Initialization, and Shutdown

This chapter explains how to configure, initialize, and shut down the Scream runtime component.

Allocating System Resources

The `SceScreamPlatformInitEx2` structure stores Scream platform initialization values. You can fill the structure with default values using the `sceScreamFillDefaultScreamPlatformInitArgsEx2()` function.

For full details on default initialization values, see the `sceScreamFillDefaultScreamPlatformInitArgsEx2()` function and the Scream and synthesizer-specific system defaults in the *Scream Library Reference* documents.

Having filled the `SceScreamPlatformInitEx2` structure with default values, the next task is to customize certain values per application specifications. Some customizations are required; others are optional.

Required Customizations

On the PlayStation®Vita and PlayStation®4 platforms, the following `SceScreamPlatformInitEx2` structure members must have custom settings:

- `SceScreamPlatformInitEx2.memAlloc`
- `SceScreamPlatformInitEx2.memFree`

On the Windows platform, *`memAlloc`* and *`memFree`* can retain the default value of `NULL`, in which case, internal memory allocation and free functions are used.

You must set the *`memAlloc`* and *`memFree`* members with the addresses of custom memory allocation and free functions, based, respectively, on the `SceScreamExternSndMemAlloc` and `SceScreamExternSndMemFree` prototypes.

Optional Customizations

While the default `SceScreamPlatformInitEx2` structure initialization values provide basic settings required to run Scream, you can also provide custom settings for any members based on the specifics of your application.

Allocating Synthesizer Voice Types

Scream on NGS and NGS2 allocates the following default number of voices types:

- 64 mono VAG voices
- 16 mono PCM voices
- 32 mono ATRAC9™ voices
- 8 stereo (two-channel) VAG voices
- 8 stereo (two-channel) PCM voices
- 8 stereo (two-channel) ATRAC9™ voices.

And on NGS2 only, Scream allocates the following default number of voices types:

- 8 5.1 (six-channel) PCM voices
- 8 7.1 (eight-channel) PCM voices
- 8 5.1 (six-channel) ATRAC9™ voices
- 8 7.1 (eight-channel) ATRAC9™ voices

You can custom set the number of voices to allocate of each respective type using the (embedded) `SceScreamSystemParams` *`voiceTypeCount`* member. The *`voiceTypeCount`* member takes an array of values, ordered per the synthesizer-specific set of Voice Data Type Constants, as shown in Table 4.

Table 4 Synthesizer-Specific Order of Voice Data Type Constants

NGS	NGS2
- Number of VAG mono voices	- Number of VAG mono voices
- Number of PCM mono voices	- Number of PCM mono voices
- Number of ATRAC9™ mono voices	- Number of ATRAC9™ mono voices
- Number of VAG stereo voices	- Number of VAG stereo voices
- Number of PCM stereo voices	- Number of PCM stereo voices
- Number of ATRAC9™ stereo voices	- Number of ATRAC9™ stereo voices
	- Number of PCM 5.1 voices
	- Number of PCM 7.1 voices
	- Number of ATRAC9™ 5.1 voices
	- Number of ATRAC9™ 7.1 voices

For example, on PlayStation®4, if you are using ATRAC9™ assets in a Bank, you might set the `SceScreamSystemParams` *`voiceTypeCount`* member to increase the allocation of various ATRAC9™ voice types, while keeping the defaults for VAG and PCM. In this case, for 16 ATRAC9™ mono, 16 ATRAC9™ stereo, 16 ATRAC9™ 5.1, and 16 ATRAC9™ 7.1 voices, you would set the *`voiceTypeCount`* member as follows:

```
SceScreamSystemParams.voiceTypeCount =
[SCE_SCREAM_SND_DEFAULT_NUM_VAG_MONO_VOICES,
SCE_SCREAM_SND_DEFAULT_NUM_PCM_MONO_VOICES,
16,
SCE_SCREAM_SND_DEFAULT_NUM_VAG_STEREO_VOICES,
SCE_SCREAM_SND_DEFAULT_NUM_PCM_STEREO_VOICES,
16,
SCE_SCREAM_SND_DEFAULT_NUM_PCM_5_1CH_VOICES,
SCE_SCREAM_SND_DEFAULT_NUM_PCM_7_1CH_VOICES,
16,
16];
```

The total number of allocated voices (of all types) may exceed the maximum number of voices that can be simultaneously addressed by Scream – as returned by the `sceScreamGetMaxPolyphony()` function. In fact, to accommodate peak usage of single voice types it may be desirable to set a total number of allocated synthesizer voices greater than the number of Scream-addressable voices. For example, if you allocate 64 VAG mono, 64 PCM mono, and 64 ATRAC9™ mono voices (a total of 192), the synthesizer could not play more than 64 voices of any one type simultaneously. Conversely, if you allocate 192 voices of each type, the synthesizer could potentially play all 192 voices of a single type simultaneously; assuming maximum polyphony is set to 192 or higher. Allocating an appropriate number of synthesizer voices of each type is therefore a game-specific tuning issue.

Setting Maximum Polyphony

Maximum polyphony is the maximum number of synthesizer voices (of any type) that can be simultaneously addressed by Scream. Maximum polyphony is the threshold beyond which the voice stealing mechanism is activated. You can set the maximum number of voices of polyphony at initialization time using the `SceScreamPlatformInitEx2` structure's *`maxPolyphony`* member. Without a setting, maximum polyphony defaults to the value of the constant `SCE_SCREAM_SND_DEFAULT_MAX_POLYPHONY` (64 on the NGS and NGS2 synths).

The approximate memory overhead for a single voice of polyphony is 120 bytes. So, for example, with default maximum polyphony the dynamic memory requirement when calling `sceScreamStartSoundSystemEx2()` is around 7.5 kB on NGS/NGS2.

You can verify Scream's maximum polyphony value using the `sceScreamGetMaxPolyphony()` function. Calling this function after initializing Scream returns the value set in the `SceScreamPlatformInitEx2` structure's *`maxPolyphony`* member. Conversely, calling `sceScreamGetMaxPolyphony()` *before* initializing Scream returns zero; as the maximum polyphony value would not yet have been set.

Allocating Pre-Master Submixes

The pre-master submix busses are a helpful feature in Scream. By routing Group or individual Sound outputs to pre-master submixes you can achieve greater control over final mix levels, as well as leverage signal processing functionality available in the pre-master submix voices. See the "Working with Pre-Master and Master Signal Processors" chapter for full details of processing units available on the pre-master submix, and master busses.

NGS and NGS2 provide two types of pre-master submix voice:

- Pre-master submix with compressor
- Pre-master submix with compressor that accepts side-chain input

Without making a custom setting, Scream allocates a default of zero pre-master submix voices of either type. However, by setting the `SceScreamSystemParams` *`numPremasterCompSubmixes`* and *`numPremasterScCompSubmixes`* members, you can allocate up to 4 pre-master submix voices (of both types).

Setting System Behaviors

The initialization flags define a variety of behaviors with which you can initialize Scream and its supported synthesizers. You set Scream initialization flags in the *`initFlags`* member of the `SceScreamPlatformInitEx2` structure. You set synthesizer initialization flags in the *`initFlags`* member of the `SceScreamSystemParams` structure, which in turn is embedded as the *`synthParams`* member of the `SceScreamPlatformInitEx2` structure.

Scream Runtime Behaviors

The only initialization flag set by default (following a call to the `sceScreamFillDefaultScreamPlatformInitArgsEx2()` function) is `SCE_SCREAM_SSS_FLAGS_RETURN_ON_BAD_PARAM`, which causes Scream functions to return immediately if a specified parameter is invalid. Another initialization flag that you can apply to the `SceScreamPlatformInitEx2` *`initFlags`* member is `SCE_SCREAM_SSS_FLAGS_HALT_ON_BAD_PARAM`. For further details, see the "Scream Initialization Flags" section in the *Scream Library Reference* documents.

Audio Output Behaviors

Note: Audio output behaviors described in this section apply to the NGS2 synthesizer only.

You specify optional audio output behaviors by applying one or more flags to the `SceScreamSystemParams` structure's *`initFlags`* member. Use the `OR` operator for multiple selections.

Custom Personal Output

The following synthesizer initialization flags allow you to access custom personal outputs on the PlayStation®4 platform:

- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PAD_SPEAKER`
  Opens pad speaker output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PERSONAL_STEREO`
  Opens personal stereo (2-channel) ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_VOIP`
  Opens VoIP headset output ports for local players.

- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_BGM`
  Opens the background music (BGM) port.

For further details see:

- Setting Sound Output Destinations in the "Working with Sounds" chapter
- Setting Group Output Destinations in the "Working with Groups" chapter
- Routing Pre-Master Submix Output Destination in the "Working with Pre-Master and Master Signal Processors" chapter
- Setting Auxiliary Buss Output Destinations in the "Working with Auxiliary Buss Effects" chapter

Choosing Initial Playback Mode

You set an initial playback mode in the *`playbackMode`* member of the `SceScreamPlatformInitEx2` structure. When filling this structure with default values prior to initializing Scream, the `sceScreamFillDefaultScreamPlatformInitArgsEx2()` function sets the *`playbackMode`* member to `SCE_SCREAM_SND_DEFAULT_PLAYBACK_MODE`, a constant which is in turn set to `SCE_SCREAM_SPEAKER_MODE_STEREO`.

If you would prefer to use an alternative playback mode, you can re-set playback mode to `SCE_SCREAM_SPEAKER_MODE_DPL2` for Dolby™ Pro Logic II surround sound as well as other values, using the `sceScreamSetPlaybackMode()` function. See Setting Playback Mode for further details.

You can also set the *`playbackMode`* member of the `SceScreamPlatformInitEx2` structure to `SCE_SCREAM_SPEAKER_MODE_BEST`. This option auto-configures Scream to choose the optimal playback mode on the host platform.

Routing TTY Output

You can create a custom debug text output function that adheres to the `SceScreamSndDebugHandler` prototype. Then, by calling the `sceScreamSetDebugHandler()` function, you can set your custom function to receive all Scream and NGS/NGS2 debug text output – diverting output that would otherwise go to a default printf-like function. This mechanism allows you to funnel all TTY output to a single location (for example, screen, log file, custom application, and so on).

It is recommended to set your application-defined `SceScreamSndDebugHandler` prior to initializing Scream, as much information is produced during the initialization process. See Obtaining Diagnostic TTY Output in the "Working with Sounds" chapter for further details on Sound TTY output.

You can also silence TTY output by setting the `SCE_SCREAM_SSS_FLAGS_SILENT` initialization flag; see Setting System Behaviors for further details.

Initialization

Before initializing Scream, you must perform the standard steps for initializing any application.

PlayStation®Vita Standard Initialization

The steps required to initialize any PlayStation®Vita application include:

- Linking with: `-lSceSysmodule_stub`
- Calling: `sceSysmoduleLoadModule( SCE_SYSMODULE_NGS );`
- Performing any additional tasks that may be required

For further details, refer to the PlayStation®Vita SDK documentation and the *NGS Library Reference*.

PlayStation®4 Standard Initialization

The steps required to initialize any PlayStation®4 application include:

- Linking with: `-lSceSysmodule_stub_weak`
- Calling: `sceSysmoduleLoadModule( SCE_SYSMODULE_NGS2 );`
- Performing any additional tasks that may be required

For further details, refer to the PlayStation®4 SDK documentation.

Initializing Scream

The main steps required to initialize Scream are:

(1) Initialize a `SceScreamPlatformInitEx2` data structure with default values by calling `sceScreamFillDefaultScreamPlatformInitArgsEx2()`.

(2) Custom set any `SceScreamPlatformInitEx2` members to suit your needs.

> **Note:** See Required Customizations above.

(3) Call `sceScreamStartSoundSystemEx2()`, referencing your `SceScreamPlatformInitEx2` data structure.

The common header file used by the sample programs (`audio_examples.h`) shows in detail the steps required to initialize Scream on the PlayStation®Vita and PlayStation®4 platforms. See Sample Programs for further details.

Here are the main steps for initializing Scream on the PlayStation®Vita and PlayStation®4 platforms:

```
// Initialize a SceScreamPlatformInitEx2 data structure
   SceScreamPlatformInitEx2 screamInit;
   sceScreamFillDefaultScreamPlatformInitArgsEx2(&screamInit);

// Custom set your memory allocation/free functions
   screamInit.memAlloc       = my_malloc;
   screamInit.memFree        = my_free;

// Initialize Scream for use by an application
   sceScreamStartSoundSystemEx2(&screamInit);
```

> **Note:** Scream uses callback functions for memory allocation and release. These are not optional as Scream provides no defaults; therefore *`my_malloc`* and *`my_free`* must refer to valid functions.

For details on the initialization parameters and values, see the sections on the `SceScreamPlatformInitEx2` data structure and `sceScreamFillDefaultScreamPlatformInitArgsEx2()` function in the *Scream Library Reference* documents.

Verifying Initialization

Having initialized Scream, you can verify that the system is running by calling the `sceScreamGetSystemRunning()` function. The function returns `TRUE` if Scream is running, and `FALSE` if Scream is not running.

Closing Scream

Shut down Scream by calling the `sceScreamStopSoundSystem()` function. This function silences all Scream voices, and releases all Scream-allocated memory. You may also need to close any modules you initialized.

4 Working with System Globals

This chapter explains global system configuration issues.

Setting Scream Parameter Validation Options

There are three options for function-level parameter validation in Scream. You specify a parameter validation option when initializing Scream, by setting the `SceScreamPlatformInitEx2` *`initFlags`* member with a choice of either or neither – but not both – of the following flags.

Setting the Return on Bad Parameter Flag

Flag: `SCE_SCREAM_SSS_FLAGS_RETURN_ON_BAD_PARAM`

Setting this flag causes a function to return immediately if any parameter is invalid. This flag is set by default when calling `sceScreamFillDefaultScreamPlatformInitArgsEx2()` to initialize the `SceScreamPlatformInitEx2` structure.

Setting the Halt on Bad Parameter Flag

Flag: `SCE_SCREAM_SSS_FLAGS_HALT_ON_BAD_PARAM`

Setting this flag causes Scream to halt immediately if any parameter is invalid.

Setting NGS/NGS2 Parameter Validation Options

You specify synthesizer parameter validation options when initializing Scream, by setting the `SceScreamSystemParams` *`initFlags`* member. There is one option for parameter validation.

Flag: `SCE_SCREAM_SND_SYNTH_INIT_FLAG_VALIDATE_PARAMS`

Setting this flag enables validation of DSP parameter arguments with respect to the NGS and NGS2 synthesizers. If out-of-range values are encountered appropriate warnings are output to the TTY.

Configuring Per-Tick Callbacks

You can initialize Scream to make an application callback on every Scream tick. Scream ticks define the update rate of the Scream Runtime, and govern the execution speed of Sound scripts. Ticks occur at a rate of 240 per second. For further details, see Tick Count later in this chapter.

To initialize Scream to make per-tick application callbacks, you must include (using the bitwise `OR` operator) the constant, `SCE_SCREAM_SSS_FLAGS_TICK_CALLBACK`, in the value specified for the `SceScreamPlatformInitEx2` structure's *`initFlags`* member. As a result, Scream invokes your custom event callback function on every Scream tick, supplying as a value for the function's *`reason`* parameter, the constant `SCE_SCREAM_SND_EVENT_CB_REASON_SCREAM_TICK`. Your custom event callback function must be registered in the `SceScreamPlatformInitEx2` *`eventCallback`* member, and must adhere to the `SceScreamSndEventCallback` prototype.

Note: Because your custom event callback function is invoked so frequently for the per-tick callback, it is critical that processing performed within this callback be kept to an absolute minimum! Otherwise, runtime processing may be interrupted or audible dropouts may occur.

For further details, see the `SceScreamSndEventCallback` type definition in the *Scream Library Reference* documents.

The `SceScreamPlatformInitEx2` *`eventCallback`* member is also used to specify a callback to be called when a pre-defined Sound event is triggered, as described in Configuring Sound Event Callbacks.

Therefore, if you have already configured a per tick callback, you can't specify a callback for a Sound event — nor the other way around.

See also: Configuring Sound Event Callbacks.

Retrieving and Setting Global System Information

You can retrieve – and in many cases, set – global system values using API functions.

Rendering Synthesizer

You can verify which rendering synthesizer you are linked with by calling the `sceScreamGetSynthName()` function.

Simultaneously Addressable Voices

The `sceScreamGetMaxPolyphony()` function retrieves the maximum number of voices that can be simultaneously played by Scream.

Allocated Synthesizer Voices

The `sceScreamGetAllocatedVoiceCountByType()` function retrieves a count of allocated voices by data type. You specify a data type to reference – ATRAC9™ mono, ATRAC9™ stereo, and so on – using the Voice Data Type constants; see the *Scream Library Reference* documents for details. The function returns a count of allocated voices of the specified data type.

Playback Mode

You can verify the current playback mode using the `sceScreamGetPlaybackMode()` function. The function returns a value representing one of the Playback Mode constants; see the *Scream Library Reference* documents for details.

You can set the Playback mode at initialization time, and at run time, although the `SCE_SCREAM_SPEAKER_MODE_BEST` mode is not available at run time. See Setting Playback Mode for details.

Tick Count

The timing resolution used for Scream control data is measured in system ticks, which occur at the rate of 240 times per second. Each tick increments an internal counter. You can retrieve the current value of the tick counter using the `sceScreamGetTick()` function.

Global Registers

Scream maintains a set of global registers (the total number of which is `SCE_SCREAM_SND_MAX_GLOBAL_REGISTERS`; currently 64). You can use global registers to communicate game engine state to Scream Sound scripts, either for direct application to parameter values or for more complex logical operations. You can retrieve the current value of a global register using the `sceScreamGetSFXGlobalReg()` function. The function takes a single argument, specifying, in a one-based index, a global register for which to retrieve the current value.

You can also set a global register value using the `sceScreamSetSFXGlobalReg()` function.

Note: Audio designers and programmers use global registers to communicate state information between the game engine and the Scream runtime. Typically, the audio team – designers and programmers – in the initial stages of development, agree on a set of global registers and their specific usage.

Stream Path Groups

Scream maintains a set of stream path groups (the total number of which is `SCE_SCREAM_SND_MAX_STREAMING_FILE_DIRECTORIES`; currently 16). Stream path groups are used to associate *Stream* Grains (created in Scream Tool) with actual directory paths. You can verify the directory path associated with a particular path group using the `sceScreamGetStreamingFileDirectory()` function. The function takes a single argument specifying the directory path you wish to verify in a zero-based index; this argument is a number between 0 and 15. It returns the directory path string for the specified stream path group.

You can also set stream path groups using the `sceScreamSetStreamingFileDirectory()` function. Note that when using this function, the directory path you specify in the *`pDirectoryString`* parameter must end with a directory separator slash.

Consultation Point: Deciding on stream directory paths is a collaboration issue between audio programmers and designers. Before setting stream path groups, be sure to agree in advance on all stream file directory paths that will be used in your game.

Setting Playback Mode

Scream on NGS supports two Playback modes. Table 5 shows the Playback modes along with their associated constants and number of discrete output channels.

Table 5 Scream on NGS Playback Modes

Playback Mode	Constant	Channels
Stereo speakers	`SCE_SCREAM_SPEAKER_MODE_STEREO`	2
Dolby™ Pro Logic II surround sound	`SCE_SCREAM_SPEAKER_MODE_DPL2`	2

Scream on NGS2 supports additional Playback modes. Table 6 shows the Playback modes, along with their associated constants and number of discrete output channels.

Table 6 Scream on NGS2 Playback Modes

Playback Mode	Constant	Channels
Stereo speakers	`SCE_SCREAM_SPEAKER_MODE_STEREO`	2
Dolby™ Pro Logic II surround sound	`SCE_SCREAM_SPEAKER_MODE_DPL2`	2
5.1 surround sound	`SCE_SCREAM_SPEAKER_MODE_5_1`	6
7.1 surround sound	`SCE_SCREAM_SPEAKER_MODE_7_1`	8

The Playback mode determines the number of channels that Sounds are panned through; it is a global Scream mode. You can set Playback mode settings both when initializing Scream and during run time. You can also retrieve the current Playback mode setting.

Initial Playback Mode Setting

For information on setting initial playback mode see Choosing Initial Playback Mode in the "Configuration, Initialization, and Shutdown" chapter.

Run Time Playback Mode Setting

During run time you can re-set the Playback mode using the `sceScreamSetPlaybackMode()` function. The `sceScreamSetPlaybackMode()` function returns the current Playback mode. So if the return value is other than the specified Playback mode, this indicates that the specified setting was not successful.

You can also retrieve the current Playback mode setting using the `sceScreamGetPlaybackMode()` function.

Note: The playback constant `SCE_SCREAM_SPEAKER_MODE_BEST` automatically chooses an optimal playback mode, but can only be used at initialization time. `SCE_SCREAM_SPEAKER_MODE_BEST` cannot be used when setting the playback mode at run time with the `sceScreamSetPlaybackMode()` function, and is never returned by the `sceScreamGetPlaybackMode()` function; the mode actually chosen is returned.

Configuring PlayStation®4 Custom Personal Outputs

Note: Applies to the NGS2 synthesizer only.

The PlayStation®4 platform supports custom personal outputs for up to four locally connected players. These outputs are:

- Pad Speaker — Output from a player-specific PlayStation®4 controller mono pad speaker.
- Personal Stereo — Output through stereo USB headphones plugged into the PlayStation®4 console.
- Personal Voice — Output through a stereo headset plugged into a player-specific PlayStation®4 controller stereo mini-jack.

In addition to these player-specific outputs, the PlayStation®4 platform also provides a background music output port.

To enable custom outputs on PlayStation®4 you must set certain flags in the `SceScreamSystemParams` structure's *`initFlags`* member to appropriately initialize the BoomRangBuss2 synth:

- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PAD_SPEAKER` — Opens pad speaker output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PERSONAL_STEREO` — Opens personal stereo ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_VOIP` — Opens VoIP headset output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_BGM` — Opens the background music port.

Also at initialization time and using members of the `SceScreamSystemParams` structure, you must specify the maximum and initial number of local players and their user IDs, as follows:

- *`maxLocalPlayers`* — Maximum number of local players that can be logged into the game at any one time. Range: 1 to 4. Defaults to 1.
- *`numLocalPlayers`* — Number of local players that are initially logged-in. Range: 1 to *`maxLocalPlayers`*. Defaults to 1.
- *`localPlayerIDs`* — An array of user IDs of currently logged-in local players.

If the number of players or specific player IDs change during game play, you can use the `sceScreamSynthUpdateLocalPlayerIDs()` function to dynamically update Scream's record of local players.

You can route signal to a custom personal output port from any of the following sources, using the corresponding functions:

- from Sounds, using the `sceScreamPlaySoundByIndexEx()` or `sceScreamPlaySoundByNameEx()` functions. For further details, see Setting Sound Output Destinations.
- from Groups, using the `sceScreamSetGroupVoiceOutputDest()` function. For further details, see Setting Group Output Destinations.
- from auxiliary busses, using the `sceScreamSetAuxBussOutputDest()` function. For further details, see Setting Auxiliary Buss Output Destinations.

- from pre-master submix busses, using the `sceScreamSynthPremasterSubmixSetOutputDest()` function. For further details, see Routing Pre-Master Submix Output Destination.

The above functions all include an *`outputDest`* parameter. You set the *`outputDest`* parameter to a combination of a local player identifier and a personal output destination, encapsulated within the `SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM` macro. For example, to route signal to the pad speaker output for player 0, you would set the *`outputDest`* parameter as follows:

```
SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM(SCE_SCREAM_SND_OUTPUT_DEST_PLAYER_0,
SCE_SCREAM_SND_OUTPUT_DEST_PERSONAL_PAD_SPEAKER)
```

Manipulating Script Speed

You can manipulate Sound script execution speed to achieve variable-speed replays of scripts by setting a script speed factor. For information on Sound scripts, see the “Script Editor” chapter of *Scream Tool Help*. You can set the global script speed factor using the `sceScreamSetScriptSpeedFactor()` function, and retrieve the current global script speed factor using the `sceScreamGetScriptSpeedFactor()` function. You can also set Group-specific script speed factors. See the Manipulating Group Script Speed section in the “Working with Groups” chapter.

The *`speedFactor`* parameter of `sceScreamSetScriptSpeedFactor()` is an execution speed multiplier and temporarily changes the Scream tick rate. Values of 1.0 indicate normal speed. Values greater than 1.0 increasingly speed up playback. Values less than 1.0 progressively slow down playback. *`speedFactor`* must be greater than 0.0, as variable speed replay cannot stand still or play backwards. While there are no other direct programmatic constraints on the *`speedFactor`* value, there are some indirect and practical constraints. For instance, it is inadvisable to slow down the tick rate for a prolonged period.

By default, variable speed replay affects time domain scripting properties only. However, when you set the `SCE_SCREAM_SND_SCRIPTSPEED_AFFECT_PITCH` flag in the `sceScreamSetScriptSpeedFactor()` function’s *`flags`* parameter, variable speed replay affects pitch, too. Pitch shift on the NGS/NGS2 synthesizers cannot exceed two octaves above the original pitch, so the effective maximum *`speedFactor`* value for the pitch domain is 4.0.

Time-based playback parameters can be speeded up by factors in excess of 4×. Note, however, that very high speeds with many simultaneous Sounds can rapidly consume processor cycles. A practical constraint at the high end of the time domain may be to keep *`speedFactor`* value less than 20.0. At the lower extremes, because *`speedFactor`* temporarily changes the Scream tick rate, very low *`speedFactor`* values could cause a noticeable delay following the replay until the tick rate returns to 240 ticks-per-second. A practical constraint at the low end may be to keep the *`speedFactor`* value greater than 0.05. If you need a slower replay speed, use pause/continue functionality to affect frame-by-frame replays.

Note also that by default envelope values (ADSR) are not affected by variable speed replays, so long attacks, decays, and releases remain the same no matter what the *`speedFactor`* is set to. However, when you set the `SND_SCRIPTSPEED_AFFECT_ADSR` flag in the `sceScreamSetScriptSpeedFactor()` function’s *`flags`* parameter, the durations of ADSR envelope segments are scaled by the inverse of the script speed factor.

In conjunction with `sceScreamSetScriptSpeedFactor()`, you can use the `sceScreamSetRandomIndex()` function to set the seed index of the Scream random number generator. The seed index can be used in relation to variable speed replays, where some audio parameter values are determined by random numbers, and there is a need for identical repetition of the original playback.

Stopping All Sounds

You can stop all Sounds on a system-wide basis using the `sceScreamStopAllSounds()` function. You can also stop Sounds on an individual, Group, or Bank basis. See the following sections for further information:

- Stopping a Sound
- Stopping All Sounds in a Group
- Stopping All Sounds in a Bank

Unlike the individual, Group, and Bank stop-sound functions, `sceScreamStopAllSounds()` provides no behavior parameter: in this sense it is considered an emergency stop button. `sceScreamStopAllSounds()` performs a *hard stop*; that is, audio generation stops almost instantaneously, depending on how far ahead the rendering synthesizer has buffered samples for playback.

The `sceScreamStopAllSounds()` function does not stop Streams. Use the Sndstream function `sceScreamStopAllStreams()` to stop all Streams.

5 Working with Banks

This chapter explains how to work with Banks.

What is a Bank?

A Bank is the basic data format used by Scream. It contains both control data (in the form of Sound scripts), and audio data (the ATRAC9™, PCM, and VAG data used in *Waveforms*).

Loading a Bank

You must load a Bank before playing any of its Sounds. You can load a Bank using either of the following methods:

- From disk: referencing a full path to a Bank (`BNK`) file; optionally, with an offset number of bytes into the file.
- From main memory: referencing a pointer to a memory address where the Bank resides.

Loading from Disk

To load a Bank from disk, use the `sceScreamBankLoadEx()` function. If a data management system is being used (in which `BNK` files are embedded into larger container files), the offset parameter provides a way to reference an embedded `BNK` file by specifying the number of bytes into the container file where the `BNK` file begins.

Note: Banks are platform-specific. Be sure that you are loading a Bank for the intended platform.

Loading from Memory

To load a Bank from main memory, you must first load the entire Bank into main memory. You then call the `sceScreamBankLoadFromMemEx()` function, referencing a pointer to the memory address where the Bank resides in the `loc` parameter.

Note: Because memory-loaded Banks are used in place, do not free the associated memory. That is, unless you first notify Scream that the Bank is no longer needed (with a call to `sceScreamUnloadBank()`).

Checking Load Errors

Both the load-from-disk and load-from-memory functions, `sceScreamBankLoadEx()` and `sceScreamBankLoadFromMemEx()`, return a `SceScreamSFXBlock2` pointer if the load operation is successful. If the load operation is not successful, they return `NULL`. In this case you can check the error condition by calling the `sceScreamGetLastLoadError()` function. The resulting error condition can either be an OS-specific file system error code or one of the following Scream-specific error codes:

- `SCE_SCREAM_SND_LOAD_ERROR_COULDNT_OPEN_FILE`:
  indicates that the Bank file could not be opened
- `SCE_SCREAM_SND_LOAD_ERROR_READING_FILE`:
  indicates that a problem occurred reading a Bank file.

Locating a Bank

Both functions for playing a Sound, `sceScreamPlaySoundByIndexEx()` and `sceScreamPlaySoundByNameEx()`, require a `SceScreamSFXBlock2` pointer as the bank argument. If you have lost track of the `SceScreamSFXBlock2` pointer for a particular Bank, you can retrieve it by reference to the Bank's name using the `sceScreamFindLoadedBankByName()` function.

Conversely, you may know a Bank's pointer but not its name. You can retrieve a Bank name by reference to its pointer using the `sceScreamFindLoadedBankNameByPointer()` function. The function stores the Bank's name in a `char` array that you pass in as the `outBankName` parameter.

Retrieving Bank Information

Three functions enable you to retrieve information about a loaded Bank:

- `sceScreamBankGetNumSoundsInBank()`
  Retrieves the number of sounds in a Bank
- `sceScreamBankGetSoundNameByIndex()`
  Retrieves the name of a sound by reference to its index within a Bank
- `sceScreamBankGetSoundIndexByName()`
  Retrieves the index of a Sound by reference to its name

Stopping All Sounds in a Bank

You can stop all Sounds belonging to a Bank using the `sceScreamStopAllSoundsInBank()` function. You can also stop Sounds on an individual, Group, or system-wide basis. See the following sections for further information:

- Stopping a Sound
- Stopping All Sounds in a Group
- Stopping All Sounds

Similar to the `sceScreamStopSound()` and `sceScreamStopAllSoundsInGroup()` functions, `sceScreamStopAllSoundsInBank()` provides a behavior parameter, offering a choice of two stop behaviors:

- `SCE_SCREAM_SND_STOP_BEHAVIOR_KEYOFF`:
  Performs a graceful stop, triggering any On Stop Marker Grain events, and issuing key-off messages to active voices with ADSR Release settings.
- `SCE_SCREAM_SND_STOP_BEHAVIOR_SILENCE`:
  Performs an instantaneous stop, not triggering On Stop Marker Grain events, or Waveform and Stream Grains with ADSR Release settings.

Unloading a Bank

Unloading Banks that are no longer needed is an important task. You use the `sceScreamUnloadBank()` function. If the Bank was loaded from disk, unloading automatically frees the Bank's allocated memory with an internal call to the registered memory free function (see the `SceScreamPlatformInitEx2` *`memFree`* member).

If the Bank was loaded from memory (using the `sceScreamBankLoadFromMemEx()` function), `sceScreamUnloadBank()` simply unregisters the Bank.

Note: When unloading a memory-loaded Bank, the application is responsible for freeing the memory occupied by the unloaded Bank.

The `sceScreamUnloadBank()` function can unload a Bank either synchronously or asynchronously. You specify which in the *`synchronous`* parameter, which defaults to synchronous unloading. With synchronous unloading, it is safe to free (or overwrite) the Bank's allocated memory by the time the function returns. However, synchronous Bank unloading can sleep the calling thread for a significant time interval, and is not recommended. Instead, asynchronous Bank unloading is recommended. With asynchronous unloading, before freeing (or overwriting) the Bank's allocated memory, you must poll the Bank's status using the `sceScreamBankIsSafeToDelete()` function until that function returns `TRUE`, indicating it is safe to do so. For further details on these functions, see the *Scream Library Reference* documents.

You can unload a Bank even if Sounds from that Bank are currently playing. In this case, Scream stops the Sounds that are playing before unloading the Bank.

6 Working with Groups

This chapter explains how to work with Groups.

Setting and Retrieving Group and Master Volumes

Assigning Sounds to Groups makes it easier to control the gain levels of Sounds collectively. Audio designers can assign Sounds to Groups in Scream Tool. As well as the commonly used Group assignments (for example, Sound FX, Music, and Dialog elements), Scream allows for up to 31 Groups, as well as the overall Group master volume.

You can retrieve current Group settings using the `sceScreamGetMasterVolume()` function:

```
float sceScreamGetMasterVolume(int32_t which);
```

The `which` parameter refers to one of the Volume Group constants; see Volume Groups in the *Scream Library Reference* documents. The returned gain level is a floating-point linear value between 0.0 (`SCE_SCREAM_SND_MIN_GAIN`) and 1.0 (`SCE_SCREAM_SND_MAX_GAIN`).

You can adjust individual Volume Group levels or the overall Group master-volume level (which scales the levels of all other Volume Groups) using the `sceScreamSetMasterVolume()` function:

```
int32_t sceScreamSetMasterVolume(int32_t which, float vol);
```

Again, the `which` parameter refers to one of the Volume Group constants. The `vol` parameter takes a floating-point linear gain level between 0.0 (`SCE_SCREAM_SND_MIN_GAIN`) and 1.0 (`SCE_SCREAM_SND_MAX_GAIN`).

Soloing and Muting Groups

You can solo and mute one or more Groups using the `sceScreamSetGroupSolo()` and `sceScreamSetGroupMute()` functions. When you solo a Group, it means that audio output from all non-soloed Groups is silenced. When you mute a Group, it means that audio output from that Group is silenced, while non-muted Groups are unaffected. The `sceScreamSetGroupSolo()` and `sceScreamSetGroupMute()` functions operate in a very similar way. In the function's *`groups`* parameter you specify one or more Groups in a bit field using the Group Flags; see Group Flags in the *Scream Library Reference* documents for details. In the respective solo/mute parameters, you specify whether to activate a solo/mute or cancel (an existing solo/mute) using a Boolean value.

Setting Group Output Destinations

You can route Group signal to a choice of output destinations: to the master buss, to one of the pre-master submix busses, or (on PlayStation®4 only) to a custom player-specific voice, pad speaker, or stereo output. You specify an output destination in the *`outputDest`* parameter of the `sceScreamSetGroupVoiceOutputDest()` function, and a target Group in the function's *`group`* parameter.

To route Group signal to the master buss you set the `sceScreamSetGroupVoiceOutputDest()` *`outputDest`* parameter to `SCE_SCREAM_SND_OUTPUT_DEST_MASTER`.

To route Group signal to a pre-master submix buss, you set the *`outputDest`* parameter to the zero-based index of the desired submix buss. See Setting NGS/NGS2 Pre-Master Submix Indices for details.

Note: Pre-master submix busses must be allocated at initialization time. You allocate pre-master submix busses using the *`numPremasterCompSubmixes`* and *`numPremasterScCompSubmixes`* members of the `SceScreamSystemParams` structure. See Allocating Pre-Master Submixes for further details.

The index of the first pre-master submix is expressed in the constant `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0`, equal to zero. Permissible pre-master submix indices therefore range from zero to (`SCE_SCREAM_SND_MAX_PREMASTER_SUBMIXES - 1`), that is, 0 to 3.

Routing to Custom Personal Outputs on PlayStation®4

To route Group signal to a custom player-specific output on PlayStation®4, you set the *`outputDest`* parameter to a combination of a local player identifier and a personal output destination, encapsulated within the `SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM` macro. For example, to route Group signal to the pad speaker output for player 0, you set the *`outputDest`* parameter as follows:

```
SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM(SCE_SCREAM_SND_OUTPUT_DEST_PLAYER_0,
SCE_SCREAM_SND_OUTPUT_DEST_PERSONAL_PAD_SPEAKER)
```

To use custom personal output destinations, you must initialize Scream's record of logged-in local players using members of the `SceScreamSystemParams` structure:

- *`maxLocalPlayers`*
  Sets the maximum number of local players that can be logged-in at any one time. Range: 1 to 4.
- *`numLocalPlayers`*
  Sets the number of local players that are currently logged-in. Range: 1 to *`maxLocalPlayers`*.
- *`localPlayerIDs`*
  An array of user IDs of currently logged-in local players.

If the number of players or specific player IDs change during game play, you can use the `sceScreamSynthUpdateLocalPlayerIDs()` function to dynamically update Scream's records.

You must also enable any custom output ports you intend to use by setting the appropriate NGS2 initialization flag in the `SceScreamSystemParams` structure's *`initFlags`* member corresponding to the personal output destination:

- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PAD_SPEAKER`
  Opens pad speaker output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PERSONAL_STEREO`
  Opens personal stereo (2-channel) ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_VOIP`
  Opens VoIP headset output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_BGM`
  Opens the background music (BGM) port.

For further details, see the *Scream Library Reference* document for the PlayStation®4 platform:

- "Sound Output Destinations" in the "Constants" chapter
- `SceScreamSystemParams` in the "NGS2 Data Structures" chapter

See also:

- Configuring PlayStation®4 Custom Personal Outputs
- Setting Sound Output Destinations
- Routing Pre-Master Submix Output Destination
- Setting Auxiliary Buss Output Destinations

Retrieving Groups Routing to a Specified Output Destination

You can retrieve the set of Groups currently routing to a specified output destination using the `sceScreamGetGroupsByOutputDest()` function. You specify an output destination against which to query for routing Groups, using one of the output destination constants. The function stores the retrieved Groups in an output variable using a bitwise combination of the corresponding Group flags. You can use the retrieved set of Groups as input to Group-based transport and other functions, including:

`sceScreamPauseGroup()`, `sceScreamContinueGroup()`, `sceScreamPauseAllSoundsInGroup()`, `sceScreamContinueAllSoundsInGroup()`, `sceScreamSetGroupSolo()`, `sceScreamSetGroupMute()`, and `sceScreamStopAllSoundsInGroup()`. For further details, see `sceScreamGetGroupsByOutputDest()` in the "Group Functions" section of the *Scream Library Reference* documents.

Setting a Distance Model for a Group

For details on setting Group distance models, see Setting a Distance Model for a Group in the "Working with Distance Models, Doppler, and Spatialization" chapter.

Setting Group Voices Ranges

You can configure the Scream voice manager to limit voice allocation for each Group to a specific range of voices. This provides a way to guarantee voice availability on a Group basis. It also provides an alternative to the default voice allocation, which has all Sounds/Groups sharing the full pool of voices available on the current synthesizer.

You use the `sceScreamSetGroupVoiceRange()` function to set a voice range for each Group.

```
void sceScreamSetGroupVoiceRange(int32_t group, int32_t min, int32_t max);
```

The *`group`* parameter specifies a Group for which to assign a voice range. The *`min`* and *`max`* parameters specify the upper and lower limits of the assigned voice range. For further details, see the *Scream Library Reference* documents.

Note: To determine the number of voices available on the rendering synthesizer, use the `sceScreamGetMaxPolyphony()` function.

Pausing Group Sounds

You can non-persistently pause all active Sounds in a Group, or persistently pause a Group as a whole. You can pause Sounds individually; see Pausing a Sound for further details. You can also pause a reverb instance; see Pausing and Continuing Effect Instance for further details.

Pausing All Sounds in a Group

To pause all Sounds in a Group (or multiple Groups), use `sceScreamPauseAllSoundsInGroup()`. The `sceScreamPauseAllSoundsInGroup()` function iterates over all active Sounds (including any Child Sounds) in the specified Groups and pauses them, in the same way as if you had paused each Sound in the Groups individually. Note that if a Sound belonging to a target Group is started after the `sceScreamPauseAllSoundsInGroup()` call, it still plays. This is because `sceScreamPauseAllSoundsInGroup()` operates only on active Sounds in the target Groups at the time of the call. It does not operate on the Groups per se, and therefore is not persistent. To continue all paused Sounds in a Group (or multiple Groups), use `sceScreamContinueAllSoundsInGroup()`. The `sceScreamContinueAllSoundsInGroup()` function continues all paused Sounds (and any Child Sounds they contain) in the target Groups.

Pausing a Group

To pause a Group (or multiple Groups), use `sceScreamPauseGroup()`. The `sceScreamPauseGroup()` function pauses one or more Groups, causing all active and inactive Sounds and Child Sounds contained in the Groups to pause. Unlike `sceScreamPauseAllSoundsInGroup()`, if a Sound belonging to a target Group is started after the `sceScreamPauseGroup()` call, it starts in a paused stated, as the Group with which the Sound is associated is already paused. Thus the `sceScreamPauseGroup()` function can be considered persistent. To continue a paused Group (or multiple Groups), use the `sceScreamContinueGroup()` function.

Note: The `sceScreamContinueGroup()` function does not continue Sounds in a Group that have been specifically paused, whether paused individually (using `sceScreamPauseSound()`) or collectively (using `sceScreamPauseAllSoundsInGroup()`). Call `sceScreamContinueSound()` or `sceScreamContinueAllSoundsInGroup()` to continue these sounds.

The Scream Groups sample program illustrates Group pause functionality.

Manipulating Group Script Speed

Similar to the global manipulation of script speed, you can also manipulate script speed on a Group-specific basis. For general information on script speed manipulation, see the Manipulating Script Speed section in the “Working with System Globals” chapter.

To set a single Group’s script speed factor, you use the `sceScreamSetGroupScriptSpeedFactor()` function. You can also retrieve the current script speed factor for a Group using the `sceScreamGetGroupScriptSpeedFactor()` function.

Stopping All Sounds in a Group

You can stop all Sounds belonging to one or more Groups using the `sceScreamStopAllSoundsInGroup()` function. You can also stop Sounds on an individual, bank, or system-wide basis. See the following sections for further information:

- Stopping a Sound
- Stopping All Sounds in a Bank
- Stopping All Sounds

Similar to the `sceScreamStopSound()` and `sceScreamStopAllSoundsInBank()` functions, the `sceScreamStopAllSoundsInGroup()` function provides a behavior parameter, offering a choice of two stop behaviors:

- `SCE_SCREAM_SND_STOP_BEHAVIOR_KEYOFF`:
  Specifies a graceful stop, triggering any On Stop Marker Grain events, and issuing key-off messages to active voices with ADSR Release settings.
- `SCE_SCREAM_SND_STOP_BEHAVIOR_SILENCE`:
  Specifies an instantaneous stop, not triggering On Stop Marker Grain events, or Waveform and Stream Grains with ADSR Release settings.

Activating a Volume Ducker

Volume ducking is a technique for reducing the volume of certain sounds in order to highlight other sounds. For example, in a sports game, the volume level of a crowd noise might be reduced during an announcement or commentary. Scream manages volume ducking on a Group basis. Active Sounds in a source Group cause a volume reduction in a target Group. In this example, the crowd noise is assigned as the target Group, and the announcement/commentary is assigned as the source Group.

In Scream, the presence of one or more active Sounds in the source Group triggers ducking on the target Group(s) according to a specified attack time and full duck amount. Similarly, the absence of any active Sounds in the source Group returns the target Groups’ volumes to their original levels according to a specified release time. Note that Sounds in the source Group do not necessarily have to contain any *Waveform* Grains for ducking to take place. The Sound just has to be active, even if it is silent.

Scream provides for up to `SCE_SCREAM_SND_MAX_DUCKERS` (32) simultaneous volume duckers. You activate a volume ducker using the `sceScreamSetMasterVolumeDucker()` function. But before doing so, you must set up ducker parameters – using the `SceScreamDuckerDef` data structure – that is referenced in the `sceScreamSetMasterVolumeDucker()` call. In the `SceScreamDuckerDef` structure, you specify the `source_group` using one of the Volume Group constants, and the `target_groups` using one or more of the Group Flags; see the *Scream Library Reference* documents for details.

In the `SceScreamDuckerDef` *`full_duck_amt`* member, you specify the volume level of the target Groups when fully ducked. Values range from 0.0 to 1.0: 0.0 is equal to a 100% volume duck (reducing the volume level of the target Group(s) to zero); 0.5 is equal to a 50% volume duck; 1.0 is equal to no volume duck at all. You specify the ducker's `attack_time` and `release_time` in fractions of a second. The maximum value for both is ten seconds (10.0). A value of zero for either may produce abrupt and undesirable ducking effects.

The Scream Groups sample program illustrates volume ducker functionality.

Retrieving Group Voice and Sound Usage

Two functions allow you to retrieve, respectively, the number of active synthesizer voices, and the number of active Sounds being used by a Group:

- `sceScreamGetActiveVoiceCountByGroup()`
- `sceScreamGetActiveSoundCountByGroup()`

Both functions take a single parameter, which is a reference to one of the Volume Group constants; see the *Scream Library Reference* documents for details.

7 Working with Sounds

This chapter explains how to manipulate individual Sounds, and how to retrieve Sound status information.

Setting Sound Output Destinations

Using the *`outputDest`* parameter of the `sceScreamPlaySoundByIndexEx()` and `sceScreamPlaySoundByNameEx()` functions, you can route Sound signal to a choice of the following output destinations:

- To the master output (the default)
- To an inherited output destination set for the Group to which the Sound is assigned
- To a pre-master submix buss
- On PlayStation®4 only, to a custom player-specific voice, pad speaker, or stereo output

Routing to master buss output is the default behavior if you do not set the *`outputDest`* parameter, or if you set it to `SCE_SCREAM_SND_OUTPUT_DEST_MASTER`. The *`outputDest`* parameter is handled identically in both `sceScreamPlaySoundByIndexEx()` and `sceScreamPlaySoundByNameEx()`.

To inherit the output destination from that of the Group to which the Sound is assigned, you set the *`outputDest`* parameter to `SCE_SCREAM_SND_OUTPUT_DEST_BY_GROUP`. See Setting Group Output Destinations for further details.

To route Sound signal to a pre-master submix buss, you set the *`outputDest`* parameter to the zero-based index of the desired submix buss. See Setting NGS/NGS2 Pre-Master Submix Indices for details. See also the Scream Submix example program for a demonstration of routing a Sound through different pre-master submix busses.

Note: Pre-master submix busses must be allocated at initialization time. You allocate pre-master submix busses using the *`numPremasterCompSubmixes`* and *`numPremasterScCompSubmixes`* members of the `SceScreamSystemParams` structure. See Allocating Pre-Master Submixes in the "Configuration, Initialization, and Shutdown" chapter for further details.

The index of the first pre-master submix is expressed in the constant `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0`, equal to zero. Permissible pre-master submix indices range from zero to `(SCE_SCREAM_SND_MAX_PREMASTER_SUBMIXES - 1)`, that is 0 to 3.

Routing to Custom Personal Outputs on PlayStation®4

To route Sound signal to a custom player-specific output on PlayStation®4, you set the *`outputDest`* parameter to a combination of a local player identifier and a personal output destination, encapsulated within the `SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM` macro. For example, to route Group signal to the pad speaker output for player 0, you set the *`outputDest`* parameter as follows:

```
SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM(SCE_SCREAM_SND_OUTPUT_DEST_PLAYER_0,
SCE_SCREAM_SND_OUTPUT_DEST_PERSONAL_PAD_SPEAKER)
```

To use custom personal output destinations, you must initialize Scream's record of logged-in local players using members of the Sce`ScreamSystemParams` structure:

- *`maxLocalPlayers`*
  Sets the maximum number of local players that can be logged-in at any one time. Range: 1 to 4.
- *`numLocalPlayers`*
  Sets the number of local players that are currently logged-in. Range: 1 to *`maxLocalPlayers`*.
- *`localPlayerIDs`*
  An array of user IDs of currently logged-in local players.

If the number of players or specific player IDs change during game play, you can use the `sceScreamSynthUpdateLocalPlayerIDs()` function to dynamically update Scream's records.

You must also enable any custom output ports you intend to use by setting the appropriate NGS2 initialization flag in the `SceScreamSystemParams` structure's *`initFlags`* member corresponding to the personal output destination:

- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PAD_SPEAKER`
  Opens pad speaker output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PERSONAL_STEREO`
  Opens personal stereo (2-channel) ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_VOIP`
  Opens VoIP headset output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_BGM`
  Opens the background music (BGM) port.

For further details, see the *Scream Library Reference* document for the PlayStation®4 platform:

- "Sound Output Destinations" in the "Constants" chapter
- `SceScreamSystemParams` in the "NGS2 Data Structures" chapter

See also:

- Configuring PlayStation®4 Custom Personal Outputs
- Setting Group Output Destinations
- Routing Pre-Master Submix Output Destination
- Setting Auxiliary Buss Output Destinations

Playing a Sound

After loading a Bank, you can play the Sounds it contains either by:

- Reference to the Sound's index within the Bank using the `sceScreamPlaySoundByIndexEx()` function
- Reference to the Sound's name within the Bank using the `sceScreamPlaySoundByNameEx()` function

Both of these functions include a *`params`* parameter. This takes a pointer to an initialized `SceScreamSoundParams` data structure containing Sound parameter settings, which should be initialized with appropriate values before playing the Sound. You can also update these parameters for a Sound with the `sceScreamSetSoundParamsEx()` function. For a discussion of `SceScreamSoundParams` parameters, see Setting Parameter Values.

In the case of `sceScreamPlaySoundByNameEx()`, if the *`bank`* parameter specification is `NULL`, the function searches through all loaded Banks, and plays the first Sound it finds with a name matching that specified in the *`name`* parameter.

Pausing a Sound

You can pause an individual Sound using the `sceScreamPauseSound()` function. You can also pause Sounds on a Group basis; see Pausing Group Sounds for details. And you can pause effect instances; see Pausing and Continuing Effect Instances for further details.

The `sceScreamPauseSound()` function takes a single argument specifying the Sound's handle. The `sceScreamPauseSound()` function pauses the specified Sound and any Child Sounds it contains. If the Sound is already paused, the function has no effect. Pausing a Sound does not free the voices it is using. To continue (unpause) a paused Sound, use the `sceScreamContinueSound()` function with the paused Sound's handle as the only argument. The `sceScreamContinueSound()` function continues a paused Sound and any Child Sounds it contains. If the Sound is not currently paused, the function has no effect.

Stopping a Sound

You can stop an individual Sound using the `sceScreamStopSound()` function. You can also stop all Sounds in a Group, Bank, or on a system-wide basis. See the following sections for further information:

- Stopping All Sounds in a Group
- Stopping All Sounds in a Bank
- Stopping All Sounds

Similar to the `sceScreamStopAllSoundsInGroup()` and `sceScreamStopAllSoundsInBank()` functions, `sceScreamStopSound()` provides a *`behavior`* parameter, offering a choice of two stop behaviors:

- `SCE_SCREAM_SND_STOP_BEHAVIOR_KEYOFF`:
  Specifies a graceful stop, triggering any On Stop Marker grain events, and issuing key-off messages to active voices with ADSR Release settings.
- `SCE_SCREAM_SND_STOP_BEHAVIOR_SILENCE`:
  Specifies an instantaneous stop, not triggering On Stop Marker grain events, or Waveform and Stream grains with ADSR Release settings.

Retrieving and Setting Sound Register Values

You can retrieve and set Sound register values individually, collectively, and atomically. You can also retrieve and set the values of global registers; see Global Registers.

Consultation Point: Audio designers use Sound registers when creating Sound scripts in Scream Tool; they function as Sound-specific variables. Consult with your audio designer before setting Sound registers, as this may impact the internal logic of a Sound script.

Note: Local registers apply to scripted Sounds, and should not be confused with local variables that are used as input to controlling CCSounds. For further details, see the Understanding a CCSound's Associated Variables section in the "Working with CCSounds" chapter.

Individually Getting/Setting Sound Registers

You can individually get and set the value of a Sound register using the `sceScreamGetSoundReg()` and `sceScreamSetSoundReg()` functions. You specify the following:

- Handle of the Sound for which to get/set the register value
- Specific register to get/set, using a one-based index from 1 to the total number of Sound-specific registers (`SCE_SCREAM_SND_MAX_REGISTERS`: 8)
- Either a variable in which to receive the register value (in the case of `sceScreamGetSoundReg()`), or the value to set (in the case of `sceScreamSetSoundReg()`)

Collectively Getting/Setting Sound Registers

You can collectively get and set the values of all registers belonging to a Sound using the `sceScreamGetAllSoundReg()` and `sceScreamSetAllSoundReg()` functions. You specify the following:

- Handle of the Sound for which to collectively get/set all register values
- Array (of length `SCE_SCREAM_SND_MAX_REGISTERS`: 8) in which to either receive the register values (in the case of `sceScreamGetAllSoundReg()`), or specify the values to set (in the case of `sceScreamSetAllSoundReg()`)

Atomically Getting/Setting Sound Registers

You can atomically get and set the values of all registers belonging to a Sound using, respectively, the `sceScreamLockAllSoundReg()` and `sceScreamUnlockAllSoundReg()` functions. You specify the following:

- Handle of the Sound for which to atomically get/set all register values
- Array (of length `SCE_SCREAM_SND_MAX_REGISTERS`: 8) in which to either receive the register values (in the case of `sceScreamLockAllSoundReg()`), or specify the values to set (in the case of `sceScreamUnlockAllSoundReg()`)

The atomic get/set register functions allow you to process the register values with the knowledge that they will not be changed by a script during processing time.

Note: In locking a Sound's local registers, `sceScreamLockAllSoundReg()` effectively locks the entire rendering synthesizer. It is therefore critical that calls to it are followed by a matching call to `sceScreamUnlockAllSoundReg()`, and that processing performed between these two calls is kept to an absolute minimum!

Retrieving and Setting Sound Parameter Values

You can retrieve and set Sound parameter values in the `SceScreamSoundParams` structure.

Retrieving Parameter Values

You can retrieve all current Sound parameter values using the `sceScreamGetSoundParamsEx()` function. You specify the handle of the Sound and a pointer to a `SceScreamSoundParams` data structure into which the retrieved parameter values are stored. Note that if the `SCE_SCREAM_SND_MASK_SYNTH_PARAMS` bit is set in the `SceScreamSoundParams` *`mask`* member, this indicates that one or more synthesizer-specific parameters have been set from the Scream API (overriding settings that may have been included in Bank contents), and that the `SceScreamSynthParams` structure in the *`synthParams`* parameter has the changed values. To verify which synthesizer-specific parameters have been set from the Scream API, examine the value of the `SceScreamSynthParams` *`mask`* member.

If the `SCE_SCREAM_SND_MASK_SYNTH_PARAMS` bit is not set, then the synthesizer parameters are as specified in the Bank contents, and have not been changed by the Scream API. In this case, the corresponding `SceScreamSoundParams` *`synthParams`* members do not contain data. Synthesizer parameters set in the Bank contents are on a per-Grain basis, and therefore cannot be represented in single `SceScreamSynthParams` data structure. For more information, see Parent and Child Sounds and Scream Tool and API Parameter Settings in the "System Overview" chapter.

Setting Parameter Values

You can set Sound parameter values using the `sceScreamSetSoundParamsEx()` function. You specify the handle of the Sound, and a pointer to a `SceScreamSoundParams` data structure containing the parameter values you wish to set.

Consultation Point: If the `SceScreamSoundParams` data structure includes `SceScreamSynthParams` settings, these settings will override all synthesizer-specific parameter settings made in the Scream Tool. This may include settings in multiple Grains, and in Child Sounds. Therefore, it is good practice to coordinate synthesizer parameter settings with your audio design team. See Scream Tool and API Parameter Settings in the "System Overview" chapter.

Parameter settings with respect to `SceScreamSoundParams` structure members made with the `sceScreamSetSoundParamsEx()` function interact with existing parameter values in various complementary ways. Table 7 explains these interactions for each parameter.

Table 7 Parameter Interactions

SceScreamSoundParams Member	Interaction with Existing Setting(s)
userCtx	Overrides any existing Sound user data values.
gain	Scales all Parent/Child Sound volumes, and all contained Waveform and Stream Grain volumes. Overall Sound gain is also scaled by other components. See Gain Components below.
azimuth	Value is added, modulo 360, to all Parent/Child Sound pan settings, and all contained *Waveform* and *Stream* Grain pan settings.
focus	No interaction with existing settings, as (currently) focus can only be set from the API.
pitchTranspose	Value (in fines) is added to all Parent/Child *Waveform* and *Stream* Grain note offset settings, and clamped within the range `-SCE_SCREAM_SND_MAX_PITCH_TRANSPOSE` to `+SCE_SCREAM_SND_MAX_PITCH_TRANSPOSE`. Overall Sound pitch transposition is also dependent on other components. See Pitch Transpose Components below.
pitchBlendFactor	Value is added to any amounts specified in the *Set Pitchbend*, *Add Pitchbend*, or *Random Pitchbend* Grains, and clamped within the range `SCE_SCREAM_SND_MIN_PITCH_BEND_FACTOR` to `SCE_SCREAM_SND_MAX_PITCH_BEND_FACTOR`. Overall Sound pitchbend factor is also dependent on other components. See Pitchbend Factor Components below.
registers	Overrides any existing Sound register values. See Retrieving and Setting Sound Register Values.

See Scream Tool and API Parameter Settings in the "System Overview" chapter for additional details.

Retrieving Sound Information and Status

You can retrieve a variety of current information and status about a Sound.

Gain Components

You can retrieve all the individual components that comprise a Sound's total instantaneous gain level using the `sceScreamGetSoundGainComponents()` function. You specify the handle of the Sound and a pointer to a `SceScreamGainComponents` structure into which the retrieved gain component values are stored. To determine a Sound's total gain level, multiply together all constituent gain component values. For example, suppose you call `sceScreamGetSoundGainComponents()` on a Sound, and the function stores the values listed in Table 8 in a `SceScreamGainComponents` structure.

Table 8 Gain Components Structure Values

SceScreamGainComponents Member	Value	Description
masterVolume	`0.75`	Master volume setting.
sfxGain	`0.8`	Original gain level set on the Sound handle by the content.
apiGain	`0.6`	Level set on the Sound handle from an API call.
lfoGain	`0.45`	Volume factor created by an active LFO.
autoGain	`1.0`	Gain factor of an automated gain change applied to the Sound.
groupGain	`0.85`	Level of the Sound's assigned Volume Group.
duckerGain	`1.0`	Volume factor created by an active ducker.
directSendGain	`1.0`	Level of the Sound's direct send gain (as set in the `SceScreamSynthParams` *directSendGain* member).

The total gain level in this case is: 0.75 × 0.8 × 0.6 × 0.45 × 0.85 = 0.1377

Notes:
- The `sceScreamGetSoundGainComponents()` function does not factor in individual Grain volumes.
- You may still hear a Sound's output even if its gain components calculation comes to zero. In such a case, the Sound's auxiliary send(s) are non-zero, causing signal to feed into an auxiliary buss, then through an effect and back into the master buss.

For a complete description of the `SceScreamGainComponents` structure and its members, see the *Scream Library Reference* documents.

Panning Azimuth Components

You can retrieve all the individual components that comprise a Sound's overall instantaneous panning azimuth using the `sceScreamGetSoundPanAzimuthComponents()` function. You specify the handle of the Sound and a pointer to a `SceScreamPanAzimuthComponents` structure into which the retrieved panning azimuth component values are stored. To determine a Sound's overall panning azimuth, add together all constituent panning azimuth values, modulo 360. For example, suppose you call `sceScreamGetSoundPanAzimuthComponents()` on a Sound, and the function stores the values listed in Table 9 in a `SceScreamPanAzimuthComponents` structure:

Table 9 Pan Azimuth Components Structure Values

SceScreamPanAzimuthComponents Member	Value	Description
sfxPanAzimuth	45	Original panning azimuth set on the Sound handle by the content.
apiPanAzimuth	0	Panning azimuth set on the Sound handle from an API call.
lfoPanAzimuth	340	Panning azimuth produced by an active LFO.
autoPanAzimuth	0	Panning azimuth offset of an automated panning azimuth change applied to the Sound.

The overall panning azimuth in this case is: 45 + 0 + 340 + 0 = 385 mod 360 = 25 degrees

Note: The `sceScreamGetSoundPanAzimuthComponents()` function does not factor in individual Grain panning azimuths.

For a complete description of the `SceScreamPanAzimuthComponents` structure and its members, see the *Scream Library Reference* documents.

Pitch Transpose Components

You can retrieve all the individual components that comprise a Sound's overall instantaneous pitch transposition using the `sceScreamGetSoundPitchTransposeComponents()` function. You specify the handle of the Sound and a pointer to a `SceScreamPitchTransposeComponents` structure into which the retrieved pitch transpose component values are stored.

Scream pitch transpose units are fines; which are 128th microtonal subdivisions of a semitone (or half-step). Therefore, there are 1536 (12 × 128) fines per octave; expressed in the constant `SCE_SCREAM_SND_FINES_PER_OCTAVE`. The nominal limit of pitch transposition, up or down, is 5 octaves; expressed in the constant `SCE_SCREAM_SND_MAX_PITCH_TRANSPOSE`, and equal to 7680 fines. However, due to the limits of human hearing, the synthesizer sampling rate, or a combination of the two, it may not always be possible to pitch transpose a Sound by the full five octaves.

To determine a Sound's overall pitch transposition, add together all constituent pitch transpose component values. For example, suppose you call `sceScreamGetSoundPitchTransposeComponents()` on a Sound, and the function stores the following fines values listed in Table 10 in a `SceScreamPitchTransposeComponents` structure.

Table 10 Pitch Transpose Components Structure Values

SceScreamPitchTransposeComponents Member	Value	Description
`sfxPitchTranpose`	`-3072`	Original pitch transposition amount set by the content. (`-3072` fines is 2 octaves down).
`apiPitchTranspose`	`256`	Pitch transposition amount set by an API call. (`256` fines is 2 semitones (a whole-step) up).
`lfoPitchTranspose`	`-64`	Pitch transposition amount created by an active LFO. (`-64` fines is half a semitone down).
`autoPitchTranspose`	`0`	Current pitch transposition amount of a `sceScreamAutoPitchTranspose()` call.

The overall pitch transpose amount in this case is: `-3072 + 256 + -64 = -2880` fines; equivalent to 1 octave, 10½ semitones down.

Pitchbend Factor Components

You can retrieve all the individual components that comprise a Sound's overall instantaneous pitchbend factor using the `sceScreamGetSoundPitchBendFactorComponents()` function. You specify the handle of the Sound and a pointer to a `SceScreamPitchBendFactorComponents` structure into which the retrieved pitchbend factor component values are stored.

Pitchbend factor alone will not bend/modify the pitch of any Sound unless a corresponding upper or lower pitchbend range has been set in one or more of the Sound's *Waveform* or *Stream* Grains. Pitchbend range, therefore, can only be set in Scream Tool.

Pitchbend factor acts as a scalar factor on any pitchbend range settings. If the pitchbend factor is greater than 1, it scales the upper pitchbend range; conversely, if the pitchbend factor is less than 1, it scales the lower pitchbend range. Without a pitchbend range setting in one or more of a Sound's *Waveform* or *Stream* Grains, no amount of pitchbend factor, regardless of its source (whether set in the content, an API call, or a LFO) will have any effect on the Sound.

To determine a Sound's overall pitchbend factor, add together all constituent pitchbend factor component values. For example, suppose you call `sceScreamGetSoundPitchBendFactorComponents()` on a Sound, and the function stores the values listed in Table 11 in a `SceScreamPitchBendFactorComponents` structure.

Table 11 Pitchbend Factor Components Structure Values

SceScreamPitchBendFactorComponents Member	Value	Description
`sfxPitchBendFactor`	`0.375`	Original pitchbend factor set by the content.
`apiPitchBendFactor`	`0.25`	Pitchbend factor set by an API call.
`lfoPitchBendFactor`	`-0.125`	Pitchbend factor created by an active LFO.
`autoPitchBendFactor`	`0`	Current pitchbend factor applied by a `sceScreamAutoPitchBend()` call.

The overall pitchbend factor amount in this case is: `0.375 + 0.25 + -0.125 = 0.5`

For example, suppose the high pitchbend range of a *Waveform* Grain contained in this Sound is set to four (that is, semitones). In this instance, the Sound plays at two (4 × 0.5) semitones up from its normal pitch.

3D Designer Parameters

You can retrieve 3D parameter data from Sounds or CCSounds, set by designers in Bank contents. Using the `sceScreamSoundIndexGet3DDesignerParams()`, `sceScreamSoundNameGet3DDesignerParams()`, and `sceScreamSoundInstanceGet3DDesignerParams()` functions, you can reference a Sound to query either by index or name within a Bank or by instance handle. Technically, 3D parameter data is not the property of Sound/CCSounds, but actually belongs to asset Grains (*Waveform*, and *Stream* Grains) contained in a Sound's script. For CCSounds, the functions retrieve 3D parameter data from asset Grains contained in the scripts of Sounds controlled by a CCSound.

The 3D parameter retrieval functions store retrieved data in an array of `SceScreamSnd3DGrainData` structures that you specify in the *`out3dGrainData`* parameter. The functions retrieve 3D parameter data from a maximum number of asset Grains that you specify in the *`maxCount`* parameter. The actual number of asset Grains from which 3D parameter data is retrieved is stored in a variable specified in the *`outNum3dGrains`* parameter. If the value of *`outNum3dGrains`* is less than the value you specified for *`maxCount`*, you can be sure that data was retrieved from all associated asset Grains. However, if *`outNum3dGrains`* is equal to the value you specified for *`maxCount`*, the retrieved 3D data may or may not represent all associated asset Grains.

See also: Distance Models in the "Working with Distance Models, Doppler, and Spatialization" chapter.

3D Runtime Components

You can retrieve 3D runtime attenuation components from Sounds or CCSounds. Using the `sceScreamSoundInstanceGet3DComponents()` function, you reference a Sound by instance handle. Technically, 3D attenuation components are not properties of a Sound, but actually belong to asset Grains (*Waveform* and *Stream* Grains) contained in a Sound's script. For CCSounds, the function retrieves 3D components from asset Grains contained in the scripts of Sounds controlled by a CCSound.

The 3D attenuation components retrieval function stores retrieved data in an array of `SceScreamSnd3DComponents` structures that you specify in the *`out3dComponents`* parameter. The function retrieves 3D attenuation components from a maximum number of asset Grains that you specify in the *`maxCount`* parameter. The actual number of asset Grains from which 3D attenuation components are retrieved is stored in a variable specified in the *`outNum3dcomponents`* parameter. If the value of *`outNum3dcomponents`* is less than the value you specified for *`maxCount`*, you can be sure that data was retrieved from all associated asset Grains. However, if *`outNum3dcomponents`* is equal to the value you specified for *`maxCount`*, the retrieved 3D components may or may not represent all associated asset Grains.

See also: Distance Models in the "Working with Distance Models, Doppler, and Spatialization" chapter.

Is Sound Still Playing?

You can verify whether a Sound is still playing using the `sceScreamSoundIsStillPlaying()` function. You specify the handle of the Sound to verify, as returned by the `sceScreamPlaySoundByIndexEx()` or `sceScreamPlaySoundByNameEx()` functions. The `sceScreamSoundIsStillPlaying()` function returns the Sound handle if the Sound is still playing, and returns 0 if the Sound has stopped.

Is Sound a Looper?

The designation *looper* is intended to differentiate looping Sounds from one-shot Sounds. The determination as to whether a Sound is a looper is based solely on a Sound's Looping property, set in Scream Tool. Audio designers can select Yes or No for the value of this property. A Sound containing a looping *Waveform* may qualify the Sound as a looper, but then the Sound's script may subsequently stop the looping Waveform. A Sound with a scripted loop may qualify the Sound as a looper, but then the script may be conditional, causing it to subsequently stop looping.

Consultation Point: Audio designers are best qualified to determine whether a Sound is a looper based on the Sound's functionality. Consult with your audio designer if you have questions regarding the looping status of a Sound.

You can retrieve a Sound's looping status, by reference to the Sound's index or name within a Bank or by its instance handle, using the `sceScreamIsSoundIndexALooper()`, `sceScreamIsSoundNameALooper()`, or `sceScreamIsSoundInstanceALooper()` functions respectively. These functions return 0 if the Sound has not been designated as a looper in the Bank, and return 1 if the Sound has been designated as a looper.

Stream Handles?

You can retrieve the number of active stream handles associated with a Sound using the `sceScreamGetNumActiveStreamHandles()` function. You specify the handle of the Sound to query, and the function returns a count of active stream handles.

You can obtain the handle of an active stream associated with a Sound using the `sceScreamGetActiveStreamHandle()` function. You specify the handle of the Sound to query, and the zero-based index of the associated stream handle to retrieve. The function returns an active stream handle, otherwise `NULL`.

In general, Stream handles can be used the same as Sound handles. For instance, you can call `sceScreamStopSound()` with a Stream handle in the *`handle`* parameter.

Is Sound a Streamer?

The designation streamer indicates that a Sound contains streaming content. You can retrieve a Sound's streaming status, by reference to the Sound's index or name within a Bank or by its instance handle, using the `sceScreamIsSoundIndexAStreamer()`, `sceScreamIsSoundNameAStreamer()`, or `sceScreamIsSoundInstanceAStreamer()` functions respectively. These functions return 0 if the Sound contains no streaming content, and return 1 if the Sound does contain streaming content.

Has *On Stop Marker* Grain?

Audio designers use *On Stop Marker* Grains to specify scripted actions that take place after a Sound has been stopped. You can verify whether a Sound includes an *On Stop Marker* grain in its script, by reference to the Sound's index or name within a Bank or by its instance handle, using the `sceScreamSoundIndexHasOnStopMarker()`, `sceScreamSoundNameHasOnStopMarker()`, or `sceScreamSoundInstanceHasOnStopMarker()` functions respectively. These functions return 0 if the Sound has no *On Stop Marker* Grain; and return 1 if the Sound has an *On Stop Marker* Grain.

Volume Group?

You can verify a Sound's Volume Group assignment, by reference to the Sound's index or name within a Bank or by its instance handle, using the `sceScreamGetSoundIndexVolumeGroup()`, `sceScreamGetSoundNameVolumeGroup()`, or `sceScreamGetSoundInstanceVolumeGroup()` functions respectively. These functions return the Volume Group ID of the Sound if successful, and -1 if not successful. See Volume Groups in the *Scream Library Reference* documents for details.

User Data?

User data values are (up to) twelve values associated with a Sound that can optionally be entered by audio designers using the Scream Tool Sound Properties panel. You can retrieve a Sound's associated user data values, by reference to the Sound's index or name within a Bank or by its instance handle, using the `sceScreamGetSoundIndexUserDataPtr()`, `sceScreamGetSoundNameUserDataPtr()`, or `sceScreamGetSoundInstanceUserDataPtr()` functions respectively. If successful, these functions return a pointer to an array of twelve `uint32_t` values representing the Sound's associated user data. If not successful, the functions return `NULL`.

Consultation Point: Sound user data values can only be retrieved if values have been set in Scream Tool. Setting user data values requires that the *Include User Data* property is set to `TRUE` in the Document Preferences dialog.

Designer Parameters?

Designer parameters are the set of critical Sound parameters that form part of a Bank's content, as exported from Scream Tool and loaded into the Scream runtime. When retrieved from a Sound, these parameter values are stored in a `SceScreamDesignerParams` structure, and consist of the following:

- *`vol`* – Sound volume as defined in the Sound's data
- *`volGroup`* – Volume Group to which the Sound is assigned as defined in the Sound's data
- *`pan`* – Sound panning azimuth as defined in the Sound's data
- *`instanceLimit`* – Instance limit count of the Sound as defined in the Sound's data

You can retrieve designer parameters from a Sound by reference to its index, name within a Bank, or instance handle using the following functions respectively:

- `sceScreamGetSoundIndexDesignerParams()`
- `sceScreamGetSoundNameDesignerParams()`
- `sceScreamGetSoundInstanceDesignerParams()`

For details of these functions, see the *Scream Library Reference* documents.

Applying Automated Changes to Parameter Values

You can apply automated changes to Sound parameter values. Automated parameter changes perform smooth changes to a parameter from its current value to a new value over a specified time. The following Sound parameters can be subjected to automated value changes using the associated functions:

• Gain	:	`sceScreamAutoGain()`
• Pan	:	`sceScreamAutoPan()`
• Pitchbend	:	`sceScreamAutoPitchBend()`
• Pitch transpose	:	`sceScreamAutoPitchTranspose()`

You can apply these functions to any Scream object with Sound handle, including `Streams` and `Substreams`. The functions have three common parameters: *`handle`*, *`timeToTarget`*, and *`behaviorFlags`*. These are explained as follows in Table 12.

Table 12 Common Parameters

Parameter	Description
`handle`	Identifies the Sound to which to apply the parameter change.
`timeToTarget`	Specifies the time taken to reach the target parameter value. Expressed in seconds.
`behaviorFlags`	Optional parameter change behaviors. One or more of the automated parameter change constants.

Each function also has one unique parameter that specifies a target parameter value to reach by the end of the process. This value is expressed in units appropriate to the changing parameter; see the *Scream Library Reference* documents for the details of each function.

The *`behaviorFlags`* parameter allows specification of a choice of optional behaviors. These are described in Table 13.

Table 13 behaviorFlags Optional Behaviors

Behavior Flag	Description
`SCE_SCREAM_SND_AUTO_STOP_AT_DESTINATION`	Specifies that, upon reaching its target parameter value, the Sound should stop.
`SCE_SCREAM_SND_AUTO_REVERT_IF_ACTIVE`	Specifies that an active automated parameter change (that is, one that has not yet reached its target value), should return to its original value at the same rate of change as it set out with. See Revert If Active below.
`SCE_SCREAM_SND_AUTO_COUNTER_CLOCKWISE`	This flag applies to the `sceScreamAutoPan()` function only. Specifies that a panning parameter change should go in a counter-clockwise direction. For example, if changing azimuth from zero to 270 degrees, the normal direction is incremental from zero, producing a clockwise motion. Setting this behavior flag reverses the direction, such that it decrements from zero (equivalent to 360 degrees) to 270 degrees, producing a counter-clockwise motion.
`SCE_SCREAM_SND_AUTO_TAKE_SHORTEST_PATH`	This flag applies to the `sceScreamAutoPan()` function only. Specifies that a panning parameter change should go in whichever direction provides the shortest path to the target.
`SCE_SCREAM_SND_AUTO_USE_SEPARATE_FACTOR`	Specifies that an automated parameter change uses an automation-specific parameter factor in the appropriate structure, rather than the default API parameter factor. With this flag set, you can set a parameter value while an automated parameter change on the same parameter is in progress. See Use Separate Factor below.

Revert If Active

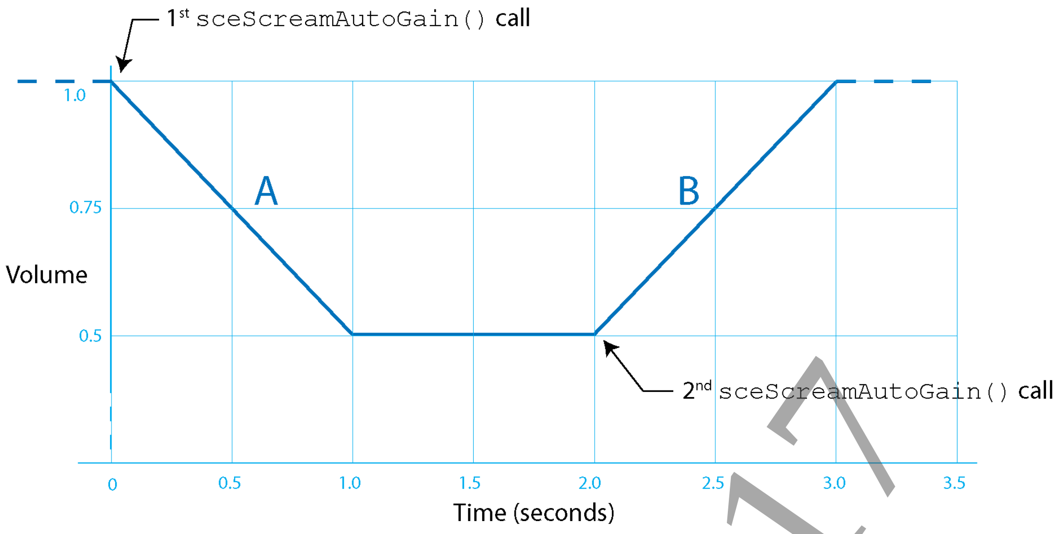
The *Revert If Active* option provides an easy way to return a parameter to its original value at the same rate of change that a similar and still-active parameter change function set out with – that is, without knowing the current parameter value at the time the second parameter change function is called.

Suppose you call `sceScreamAutoGain()` on a Sound that is currently running an automated gain change (that is, on a Sound with the same handle and where another automated gain change command has not yet completed). In this case, if you try to return to the original gain value by simply reversing the process (using the same parameter values for the second `sceScreamAutoGain()` call, except setting `targetGain` to the original value), you will get a different rate of change with the second function call. This is because the difference in gain value is less for the second call (as the first call did not arrive at its target value) but its `timeToTarget` value is the same. In such a scenario, setting the second `sceScreamAutoGain()` call's *`behaviorFlags`* parameter to `SCE_SCREAM_SND_AUTO_REVERT_IF_ACTIVE` cancels the first call, and causes the gain parameter to return to its original value at the same rate of change as it set out (with the first call).

Some examples will clarify the *Revert If Active* mechanism.

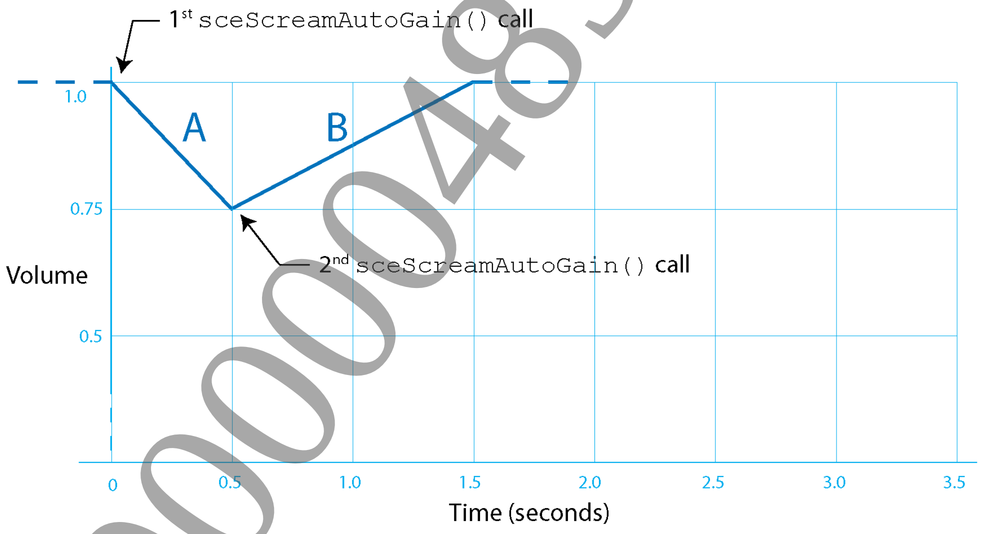
In Example 1, a Sound is playing with a gain setting of 1.0. You call the `sceScreamAutoGain()` function with `targetGain` set at 0.5, and `timeToTarget` set at one second. Two seconds later you call the `sceScreamAutoGain()` function again, this time with `targetGain` set at 1.0, and `timeToTarget` again set at one second. The resulting gain graph is shown in Figure 6.

Figure 6 Gain Graph: Example 1



In Example 2, a Sound is again playing with a gain setting of 1.0. Again, you call the `sceScreamAutoGain()` function with `targetGain` set at 0.5, and `timeToTarget` set at one second. But this time, only a half-second later you call the `sceScreamAutoGain()` function a second time, setting `targetGain` to 1.0, and `timeToTarget` again to one second. This time the second call happens before the first call has completed. The resulting gain graph is shown in Figure 7.

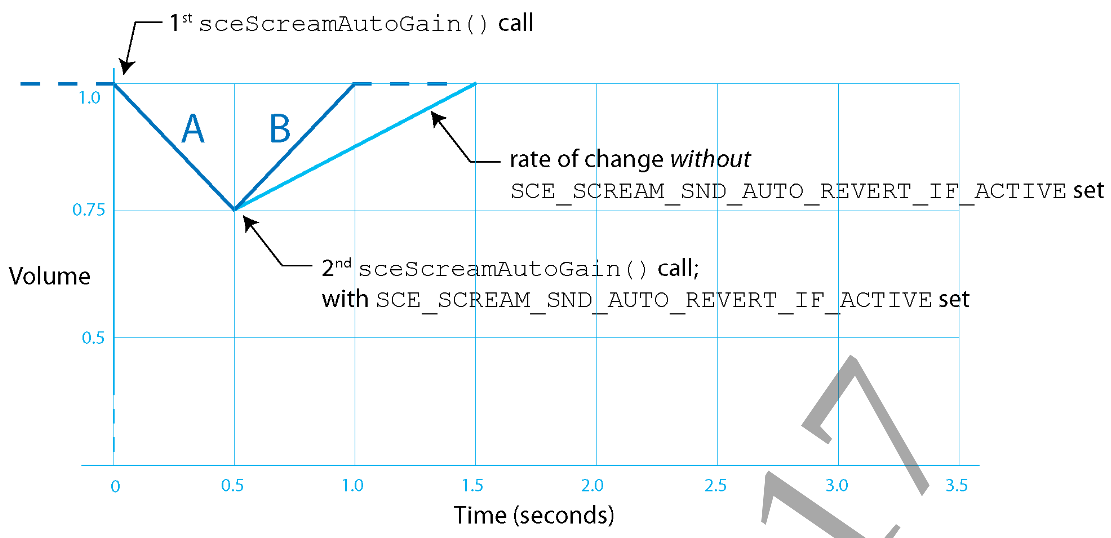
Figure 7 Gain Graph: Example 2



Notice the line segments labeled A and B in the two example graphs (Figure 6 and Figure 7). The only difference between the two examples is the time between the two `sceScreamAutoGain()` calls. In Example 1, line segment B shows a gain increase of +0.5 over 1 second. In Example 2, line segment B shows a gain increase of +0.25 over 1 second. That is, between these two examples, there is a different rate of change.

Finally, in Example 3, a Sound is again playing with a gain setting of 1.0. Again, you call `sceScreamAutoGain()` with `targetGain` set at 0.5, and `timeToTarget` set at one second. And again, a half-second later you call `sceScreamAutoGain()`, setting `targetGain` to1.0, and `timeToTarget` to one second. Only this time you also set `behaviorFlags` to `SCE_SCREAM_SND_AUTO_REVERT_IF_ACTIVE`. The resulting gain graph is shown in Figure 8.

Figure 8 Gain Graph: Example 3



This time line segments A and B show the same rate of change.

Conclusion: If you want to ensure a constant rate of change without needing to explicitly check whether a Sound is still processing an automated change to the same parameter, and without knowing its current value, set the `SCE_SCREAM_SND_AUTO_REVERT_IF_ACTIVE` behavior flag.

Use Separate Factor

As shown previously, the Scream API provides corresponding data structures for each parameter to which an automated change can be applied:

- Gain – `SceScreamGainComponents`, as discussed in Gain Components
- Panning azimuth – `SceScreamPanAzimuthComponents`, as discussed in Panning Azimuth Components
- Pitch transposition – `SceScreamPitchTransposeComponents`, as discussed in Pitch Transpose Components
- Pitchbend factor – `SceScreamPitchBendFactorComponents`, as discussed in Pitchbend Factor Components

These structures store values for all factors associated with their respective parameter. See the *Scream Library Reference* documents for details.

By default, Scream targets the default API parameter factor for automated parameter change calculations. For example, in the case of a gain parameter, Scream targets the `apiGain` factor in `SceScreamGainComponents`, and in the case of a panning azimuth parameter, Scream targets the `apiPanAzimuth` factor in `SceScreamPanAzimuthComponents`. When you set a parameter to a new value using the `sceScreamSetSoundParamsEx()` function, you are in effect also targeting the API parameter factor. Consequently, setting a parameter with the `sceScreamSetSoundParamsEx()` function while a corresponding automated parameter change is in progress would interfere with the automated process. Therefore, in this scenario, an automated parameter change process terminates if the corresponding parameter is set using the `sceScreamSetSoundParamsEx()` function.

Alternatively, by setting the `SCE_SCREAM_SND_AUTO_USE_SEPARATE_FACTOR` flag, you can instruct Scream to target an automation-specific factor (rather than the API factor) for automated parameter change calculations. For example, in the case of a gain parameter, Scream would target the `autoGain` factor in `SceScreamGainComponents`, and in the case of a panning azimuth parameter, Scream would target the `autoPanAzimuth` factor in `SceScreamPanAzimuthComponents`. Because the automated parameter change then addresses a different factor from the API parameter setting factor, you can set a parameter (using the `sceScreamSetSoundParamsEx()` function) while an automated change on the same parameter is in progress without interfering with it.

Initializing and Controlling Sound LFOs

You can initialize and control Sound LFOs (low frequency oscillators) with the `sceScreamSetSoundInstanceLFO()` function. To specify LFO parameters, you use the `SceScreamLFOParameters` structure referenced in `sceScreamSetSoundInstanceLFO()`. The number of LFOs available on a Sound instance is defined by the Scream system constant `SCE_SCREAM_SND_MAX_LFOS_PER_INSTANCE`, equal to 4. You specify the LFO you want to manipulate in the *`whichLFO`* member using a one-based index. In the *`targetParam`* member, you specify a target Sound parameter to modulate with the LFO, using one of the following LFO target constants described in Table 14.

Table 14 LFO Target Constants

LFO Target Constant	Description
`SCE_SCREAM_SND_LFO_TARGET_NONE`	No LFO target.
`SCE_SCREAM_SND_LFO_TARGET_VOL`	LFO target is a Sound's volume parameter.
`SCE_SCREAM_SND_LFO_TARGET_PAN`	LFO target is a Sound's pan (azimuth) parameter.
`SCE_SCREAM_SND_LFO_TARGET_PITCH`	LFO target is a Sound's pitch parameter.
`SCE_SCREAM_SND_LFO_TARGET_PITCHBEND`	LFO target is a Sound's pitchbend parameter.
`SCE_SCREAM_SND_LFO_TARGET_LFO_RATE`	LFO target is the rate parameter on another LFO within the same Sound instance.
`SCE_SCREAM_SND_LFO_TARGET_LFO_DEPTH`	LFO target is the depth parameter on another LFO within the same Sound instance.

The last two LFO target constants in the above table allow you to modulate the rate or depth parameter on another LFO (within the same Sound instance). In these cases, you specify the target LFO in the `SceScreamLFOParameters` *`targetLFO`* member, also using a one-based index.

Note: When using an LFO to modulate the rate or depth parameter of another LFO within the same Sound instance, you must ensure that the LFO you specify in the `SceScreamLFOParameters` *`targetLFO`* member is not the same LFO that you are setting in the `SceScreamLFOParameters` *`whichLFO`* member.

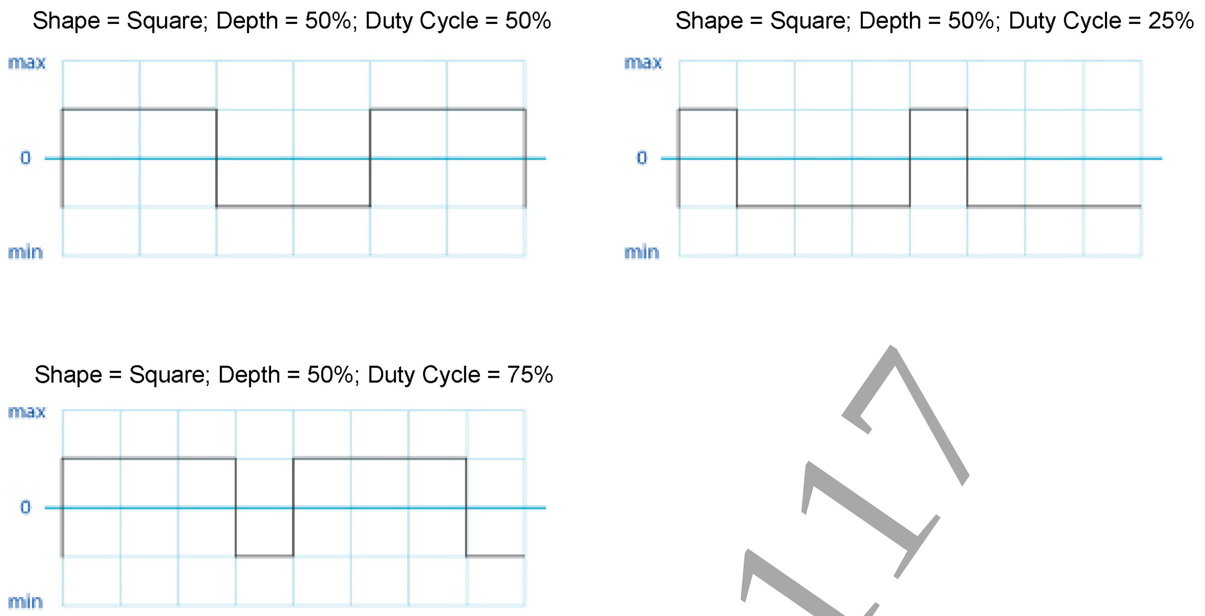
You can specify optional LFO behaviors in the *`setupFlags`* member. Here you can set one or more of the following LFO setup flags:

- `SCE_SCREAM_SND_LFO_SETUP_FLAG_INVERT` – specifies that the LFO shape is inverted.
- `SCE_SCREAM_SND_LFO_SETUP_FLAG_RAND_START_OFFSET` – specifies that the LFO shape has a random start offset.

You specify the LFO waveform shape and depth using the *`shape`* and *`depth`* members. The *`shape`* member takes one of the LFO shape constants, allowing you to specify shapes such sine, square, triangle, sawtooth, and random waves. Depth is expressed as a percentage of the range of the target parameter. For example, if you are setting a volume parameter (range: 0.0 to 1.0, with a base setting of 0.5), and you set the *`depth`* member to 50%, the LFO will modulate the volume parameter between 0.25 and 0.75. Similarly, if you if you are setting a pan parameter (range: 0 to 359), and you set the *`depth`* member to 10%, the LFO will modulate the pan parameter ±36 degrees on either side of the base setting.

The *`dutyCycle`* member applies only when the LFO shape is set to a square wave. Square waves produce only two values, one positive, one negative. Duty cycle is expressed as a percentage, with a default value of 50%. It defines the proportion of the waveform's positive state. By way of an illustration, the following diagrams in Figure 9 show a square wave with different duty cycle values.

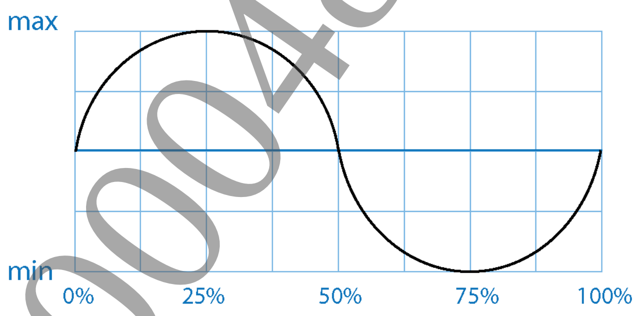
Figure 9 LFO Square Wave Duty Cycle Settings



Extreme high and low `dutyCycle` values make the square wave tend towards a pulse wave.

You can specify a start offset into the LFO shape, expressed as a percentage of one cycle of the shape, using the `startOffset` member. For example, if shape is set to a sine wave, and you set `startOffset` to 25%, the LFO will start the waveform at its maximum (rather than center) value, as shown in Figure 10.

Figure 10 LFO startOffset Expressed as a Percentage of one Cycle of the Specified Wave Shape



You specify the rate (or frequency) of the LFO using the `rate` member. Values are expressed in Hertz (cycles per second), with a range of 0.0 to 50.0.

In the `paramMask` member, you specify which members of the `SceScreamLFOParameters` structure have valid settings, using one or more of the LFO parameter mask constants. For example, if you are simply modifying the frequency of an existing LFO, you need only specify the LFO you want to address (in the `whichLFO` member), an updated frequency value (in the `rate` member), and set `paramMask` to `SCE_SCREAM_SND_LFO_MASK_RATE`. However, if you are initializing an LFO on a Sound, you must specify the full set of LFO parameters, setting `paramMask` to:

```
(SCE_SCREAM_SND_LFO_MASK_TARGET_PARAM | SCE_SCREAM_SND_LFO_MASK_SHAPE |
SCE_SCREAM_SND_LFO_MASK_RATE | SCE_SCREAM_SND_LFO_MASK_DEPTH)
```

For further details about LFO parameter masks, see "LFO Constants" in the *Scream Library Reference* documents.

To initialize or modify an LFO, you use the `sceScreamSetSoundInstanceLFO()` function. You specify the handle of a Sound upon which to apply LFO settings, and point to an appropriately initialized

`SceScreamLFOParameters` structure. The function either initializes a new LFO on the specified Sound instance or, if the specified LFO index (`SceScreamLFOParameters.`*`whichLFO`*) already exists, the function updates its parameters.

Note: When modifying the parameters of a running LFO (whether set up from Bank contents or from the Scream API), only changes to rate and depth can be performed seamlessly. Changes to all other parameters (for example, *`shape`*, *`targetParam`*, *`startOffset`*, and so on) cause the LFO to restart from the beginning of its shape.

Retrieving Sound Voice Count

You can retrieve the count of voices being used by a Sound using the `sceScreamGetSoundVoiceCount()` function. You specify the Sound's instance handle and a pointer to a `uint32_t` variable in which to receive the voice count.

Obtaining Diagnostic TTY Output

You can obtain diagnostic TTY output for a specific Sound instance or for all active Sound instances using, respectively, the `sceScreamOutputHandlerInfoToTTY()` and `sceScreamOutputAllPlayingSoundInfoToTTY()` functions. Both functions accept any combination of the TTY Output Flags, specifying a range of information types – such as tick count, Group association, gain, azimuth, focus, and so on – to be included in the TTY output. For details on the TTY Output Flags, see "TTY Output Flags" in the *Scream Library Reference* documents.

Configuring Sound Event Callbacks

Scream provides an event callback mechanism allowing you to set up custom processing when a pre-defined Sound event is triggered. Your custom event callback function is invoked whenever an event takes place for which a given Sound instance has requested a callback. The Sound handle, along with a user-specified context value and a reason value are passed to the client.

Your custom event callback function must be registered in the `SceScreamPlatformInitEx2` *`eventCallback`* member, and must adhere to the `SceScreamSndEventCallback` prototype. You set the callback condition in the `SceScreamSoundParams` *`flags`* member. Currently the only callback related option is `SCE_SCREAM_SND_FLAG_DO_FINISHED_CALLBACK`, which triggers a callback when a Sound is about to finish; additional options will be added in a future release. You can also set a context value in the `SceScreamSoundParams` *`userCtx`* member.

Note: Because your custom event callback function can be called from the synthesizer thread, it is critical that processing performed within this callback be kept to an absolute minimum! Otherwise, audible dropouts may occur.

For further details, see the `SceScreamSndEventCallback` type definition in the *Scream Library Reference* documents.

The `SceScreamPlatformInitEx2` *`eventCallback`* member is also used to specify a callback to be called every Scream tick, as described in Configuring Per-Tick Callbacks. Therefore, if you have already configured a per tick callback, you can't specify a callback for a Sound event — nor the other way around.

Setting a Sound to Pre-buffer Streaming Content

As a means of preventing possible delays in playing streaming content associated with a Sound, an optional behavior pre-buffers the stream associated with the first Stream grain contained in the Sound's script. When you set the `SCE_SCREAM_SND_FLAG_START_STREAM_PAUSED` option in the `SceScreamSoundParams` *`flags`* member, Scream locates the first Stream grain (if any), and immediately starts the stream in a paused state – poised for instantaneous playback

8 Working with Pre- and Post-Send Filters and Effects

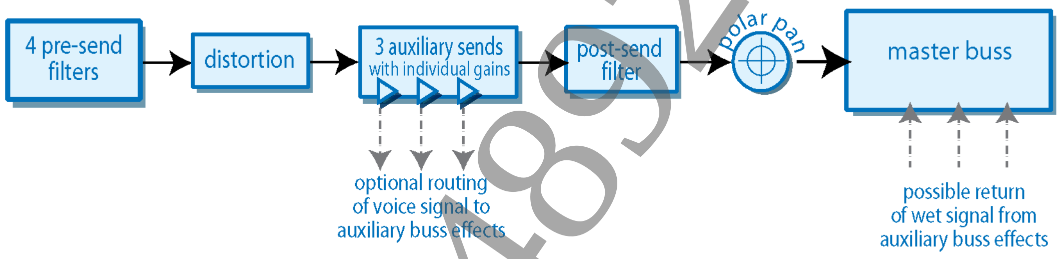
The NGS and NGS2 synthesizers provide an array of pre- and post-send filters and effects on an input voice. This chapter gives an overview, and explains how to make effective use of them from the Scream API.

Understanding Inline Filter and Effects Signal Flow

The NGS and NGS2 synthesizer input voices used by Scream Sounds, *NGS Sampler Fat Mono* and *NGS2 Sampler Fat Mono*, provide an inline series of four pre-send filters, and a pre-send distortion effect. The *NGS Sampler Fat Mono* and *NGS2 Sampler Fat Mono* voices also have three fixed auxiliary sends that can be routed to auxiliary buss effects. This architecture is similar to that of a conventional audio mixing board, where incoming signals on different channels are first equalized (EQ), then routed through auxiliary sends to an external effect. In Scream, this allows the audio characteristics of each Sound to be fine-tuned, individually, before its signal is sent to buss effects (such as reverb), where it is processed, collectively, along with signal from other Sounds sent to the same effect.

Figure 11 depicts Sound signal flow through pre-send filters and a distortion effect and a post-send filter, in relation to the auxiliary sends and master buss.

Figure 11 Pre- and Post-Send Filters and Effects Signal Flow


4 pre-send filters
distortion
3 auxiliary sends with individual gains
post-send filter
polar pan
master buss
optional routing of voice signal to auxiliary buss effects
possible return of wet signal from auxiliary buss effects

Configuring Pre-Send and Post-Send Filters

A Scream Sound, assigned to the *NGS Sampler Fat Mono* or *NGS2 Sampler Fat Mono* voice types, has four pre-send and one post-end biquad filter modules. Each filter can operate in a choice of eight filter modes (also known as filter types). A biquad filter is an infinite impulse response (IIR) filter with a rolloff of 12 dB per octave.

These filters modify incoming signal in different ways, providing a variety of tools with which you can shape the spectrum of individual Sounds. Further, the precise audio characteristics of each filter can vary considerably depending on its parameter settings.

From the API, you specify filter settings using the `SceScreamSndIIRFilterParams` structure, selecting a filter type, and assigning values to its parameters. `SceScreamSndIIRFilterParams` is embedded in the `SceScreamSynthParams` structure, which in turn is embedded in the `SceScreamSoundParams` structure. You use the `SceScreamSndIIRFilterParams` structure when setting or retrieving pre- and post-send filter values with the `sceScreamSetSoundParamsEx()` and `sceScreamGetSoundParamsEx()` functions. The eight pre-send filter types share a common set of four parameters, but the precise meaning of the parameters changes according to the filter type, and with some types, the *`gain`* parameter is not applicable.

Consultation Point: Setting pre-send filters from the Scream API overrides any settings arising from Bank contents. This applies both to filter settings within any Grain of a Sound and in any Child Sounds. Consult with your audio designer before overriding Bank settings. See Scream Tool and API Parameter Settings for details.

Available Filter Types and their Parameters

The following sections list the available filter types, explaining their parameters (that is, the `SceScreamSndIIRFilterParams` structure members as they apply to the different filter types), value ranges, and their audio characteristics. You specify the *`gain`* and *`resonance`* parameters in arbitrary units on a linear scale — *`gain`* is not specified in dB. You specify the *`cutoff`* parameter as a frequency value in Hertz.

Note: The Scream Filter sample program may be useful as a source code reference.

Bypass Filter

Bypasses all filter types. No filtering takes place. This is the default filter setting.

Table 15 Bypass Filter

Parameter	Value/Range
type	SCE_SCREAM_FLT_BQ_OFF

Low-Pass Filter

A low-pass filter attenuates frequency content above a specified cut-off frequency, allowing lower frequency content to pass. Resonance intensifies spectral content around the cut-off frequency.

Table 16 Low-Pass Filter

Parameter	Value/Range
Type	SCE_SCREAM_FLT_BQ_LPF
gain	Not applicable.
cutoff	Cut-off frequency. Range: 10.0 to 23999.0.
resonance	Resonance (or Q). Range: SCE_SCREAM_BQ_MIN_RESONANCE (0.5) to SCE_SCREAM_BQ_MAX_RESONANCE (10.0).

High-Pass Filter

A high-pass filter attenuates frequency content below a specified cut-off frequency, allowing higher frequency content to pass. Resonance intensifies spectral content at the cut-off frequency.

Table 17 High-Pass Filter

Parameter	Value/Range
type	SCE_SCREAM_FLT_BQ_HPF
gain	Not applicable.
cutoff	Cut-off frequency. Range: 10.0 to 23999.0.
resonance	Resonance (or Q). Range: SCE_SCREAM_BQ_MIN_RESONANCE (0.5) to SCE_SCREAM_BQ_MAX_RESONANCE (10.0).

All-Pass Filter

An all-pass filter allows content of all frequencies through, but alters the phase relationships of frequencies within a range specified by a center frequency and bandwidth. By combining its output with the original signal, you can create the characteristics of a comb filter.

Table 18 All-Pass Filter

Parameter	Value/Range
type	`SCE_SCREAM_FLT_BQ_APF`
gain	Not applicable.
cutoff	Cut-off frequency. Range: 10.0 to 23999.0.
resonance	Resonance (or Q). Range: `SCE_SCREAM_BQ_MIN_RESONANCE` (0.5) to `SCE_SCREAM_BQ_MAX_RESONANCE` (10.0).

Band-Pass Filter

A band-pass filter attenuates frequency content both above and below a range specified by a center frequency and bandwidth, allowing frequency content within the range to pass.

Table 19 Band-Pass Filter

Parameter	Value/Range
type	`SCE_SCREAM_FLT_BQ_BPF`
gain	Not applicable.
cutoff	Center frequency. Range: 10.0 to 23999.0.
resonance	Bandwidth of frequencies. Range: `SCE_SCREAM_BQ_MIN_RESONANCE` (0.5) to `SCE_SCREAM_BQ_MAX_RESONANCE` (10.0).

Notch Filter

A notch filter attenuates frequency content within a range specified by a center frequency and bandwidth, allowing frequency content outside the range to pass.

Table 20 Notch Filter

Parameter	Value/Range
type	`SCE_SCREAM_FLT_BQ_NOTCH`
gain	Not applicable.
cutoff	Center frequency. Range: 10.0 to 23999.0.
resonance	Bandwidth of frequencies. Range: `SCE_SCREAM_BQ_MIN_RESONANCE` (0.5) to `SCE_SCREAM_BQ_MAX_RESONANCE` (10.0).

Peaking EQ Filter

A peaking EQ filter allows cut (attenuation) or boost (intensification) of a range of frequencies, specified by a center and bandwidth of the frequency range and amount of cut/boost.

Table 21 Peaking EQ Filter

Parameter	Value/Range
type	`SCE_SCREAM_FLT_BQ_PEQ`
gain	Cut/boost. Range: `SCE_SCREAM_BQ_MIN_GAIN` (0.0) to `SCE_SCREAM_BQ_MAX_GAIN` (32.0). To specify a cut, use gain values of less than 1.0.
cutoff	Center frequency. Range: 10.0 to 23999.0.
resonance	Bandwidth of frequencies. Range: `SCE_SCREAM_BQ_MIN_RESONANCE` (0.5) to `SCE_SCREAM_BQ_MAX_RESONANCE` (10.0).

Low-Shelf Filter

A low-shelf filter attenuates or boosts frequency content below a cut-off frequency. Attenuation (or boost) is linear for all frequencies below the cut-off. The gain parameter controls attenuation (or boost) amount. Resonance controls the slope of the shelf at the cut-off point. For example, with gain=2.0 and cutoff=200.0, a low-shelf filter evenly boosts frequency content from zero to 200 Hz, while frequencies from 200 Hz to the top of the spectrum remain at unity gain, producing an effect similar to the 'loudness' button on older hi-fi systems.

Table 22 Low-Shelf Filter

Parameter	Value/Range
type	`SCE_SCREAM_FLT_BQ_LSH`
gain	Linear cut/boost. Range: `SCE_SCREAM_BQ_MIN_GAIN` (0.0) to `SCE_SCREAM_BQ_MAX_GAIN` (32.0). To specify a cut, use gain values of less than 1.0.
cutoff	Cut-off frequency. Range: 10.0 to 23999.0.
resonance	Bandwidth. Range: `SCE_SCREAM_BQ_MIN_RESONANCE` (0.5) to `SCE_SCREAM_BQ_MAX_RESONANCE` (10.0).

High Shelf Filter

A high shelf filter attenuates or boosts frequency content above a cut-off frequency. Attenuation (or boost) is linear for all frequencies above the cut-off. The gain parameter controls attenuation/boost amount. Resonance controls the slope of the shelf at the cut-off point.

Table 23 High Shelf Filter

Parameter	Value/Range
type	`SCE_SCREAM_FLT_BQ_HSH`
gain	Linear cut/boost. Range: `SCE_SCREAM_BQ_MIN_GAIN` (0.0) to `SCE_SCREAM_BQ_MAX_GAIN` (32.0). To specify a cut, use gain values of less than 1.0.
cutoff	Cut-off frequency. Range: 10.0 to 23999.0.
resonance	Bandwidth. Range: `SCE_SCREAM_BQ_MIN_RESONANCE` (0.5) to `SCE_SCREAM_BQ_MAX_RESONANCE` (10.0).

Configuring the Distortion Effect

You set distortion effect parameters using the `SceScreamSndDistortionParams` structure. `SceScreamSndDistortionParams` is embedded in the `SceScreamSynthParams` structure, which in turn is embedded in the `SceScreamSoundParams` structure. The `SceScreamSndDistortionParams` structure has members for each of the distortion effect parameters, described as follows.

Table 24 Distortion Effect Parameter Members

Member	Description
a	Distortion coefficient A. Range: 0 to 10.0.
b	Distortion coefficient B. Range: 0 to 10.0.
clip	Limiter on the audio output of the polynomial stage. Range: 0 to 1.0.
gate	Noise gate on the audio output. Range: 0 to 1.0.
wetGain	Wet (distorted signal) gain factor. Range: 0 to 4.0; where 1.0 is unity gain.
dryGain	Dry (original signal) gain factor. Range: 0 to 4.0; where 1.0 is unity gain.
flags	The only acceptable flag is `SCE_SCREAM_SND_DISTORTION_ENABLED`, which must be set to switch the distortion effect on.

Consultation Point: As with the pre-send filters, setting the distortion effect from the Scream API overrides any settings arising from Bank contents. This applies both to distortion settings within any Grain of a Sound, or in any Child Sounds. Consult with your audio designer before overriding Bank settings. See Scream Tool and API Parameter Settings for details.

In practice, distortion settings are probably best left to the designer specifications.

Applying Pre-Send, Distortion, and Post-Send Filter Settings

Whether you are applying filter settings upon starting a Sound or updating them while a Sound is playing, the corresponding functions, `sceScreamPlaySoundByIndexEx()`, `sceScreamPlaySoundByNameEx()`, and `sceScreamSetSoundParamsEx()`, all include a *params* parameter. This parameter points to a `SceScreamSoundParams` structure containing the parameter values to set from the API. Filter parameter values are stored in a `SceScreamSynthParams` structure, which is embedded in the `SceScreamSoundParams` structure in its *synthParams* member.

The `SceScreamSynthParams` structure includes five members that point to `SceScreamSndIIRFilterParams` structures: one for each of the four pre-send filter modules and one for the post-send filter module. The `SceScreamSynthParams` structure also includes a member that points to a `SceScreamSndDistortionParams` structure to specify the distortion effect.

Consultation Point: Filter settings applied from the API override any initial settings on the same Sound arising from Bank contents. Because you cannot address individual Grains within a Sound (or child Sounds), API settings at once override any and all initial filter settings. Consult with your audio designer before overriding settings in Bank contents. See Scream Tool and API Parameter Settings for details.

When applying filter settings, be sure to set the `SceScreamSynthParams` *mask* member appropriately. The *mask* member takes one or more of the Synthesizer Constants that begin with `SCE_SCREAM_SND_MASK_*` – allowing you to specify exactly which of the remaining `SceScreamSynthParams` structure members have settings and which do not. `SceScreamSynthParams` structure members not referenced in the *mask* member are not set; this will not accidentally override Scream Tool settings. For example, if you are calling `sceScreamSetSoundParamsEx()` to simply update post-send filter settings on a Sound, set the `SceScreamSynthParams` *mask* member to `SCE_SCREAM_SND_MASK_POSTSEND_FILTER`. For further details, see “Synth Parameter Bit Masks” in the *Scream Library Reference* documents.

9 Working with Auxiliary Buss Effects

This chapter explains how to work with auxiliary buss effects available on the NGS2 and NGS synthesizers.

Auxiliary Buss Effects and their Applicable Functions

Auxiliary buss effects vary according to synthesizer. On the NGS synthesizer, the only available auxiliary buss effect is the I3DL2 reverb. You instantiate reverb effects at initialization time, and use the Reverb Functions and Buss Configuration Functions to manipulate them.

On the NGS2 synthesizer, there are two alternative behaviors regarding auxiliary effects. You can either choose legacy behavior, identical to NGS, in which you instantiate reverb effects at initialization time, and use the Reverb Functions and Buss Configuration Functions to manipulate them. Or you can choose to take advantage of three auxiliary effect types (I3DL2 reverb, delay, and chorus), with dynamic, run-time instantiation, and use the Effect Functions and Effect Preset Functions to manipulate them. The choice of whether to use legacy behavior or dynamic instantiation behavior is determined by a flag set in the `SceScreamSystemParams` *`initFlags`* member when initializing Scream:

- To limit your application to legacy (reverb only) auxiliary effect behavior, do *not* set the `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API` flag.
- To take advantage of all three effect types, dynamic instantiation, and use of the effect APIs, you must set the `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API` flag.

Auxiliary buss effects for the NGS2 and NGS synthesizers and the functions by which you manipulate them are shown in the following table.

Table 25 Auxiliary Buss Effects

Synthesizer	Auxiliary Buss Effects	Effect Instantiation	Applicable Functions
NGS	I3DL2 reverb effect	Initialization time instantiation	- Reverb Functions - Buss Configuration Functions - Auxiliary Buss Functions
NGS2	I3DL2 reverb effect	Initialization time instantiation	- Reverb Functions - Buss Configuration Functions - Auxiliary Buss Functions **Note**: Do not set initialization flag `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API`
NGS2	- I3DL2 reverb effect - Delay effect - Chorus effect	Dynamic, run time instantiation	- Effect Functions - Effect Preset Functions - Auxiliary Buss Functions **Note**: Set initialization flag `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API`

These functions are further described below.

Reverb Functions

Note: The reverb functions are applicable to I3DL2 reverb instances created at initialization time only. To use these functions on the NGS2 synthesizer, do not set the `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API` initialization flag.

You can use the reverb functions to:

- Obtain handles (`SceScreamReverbHandle`) for reverb instances (`sceScreamReverbGetHandleByBuss()`). See Instantiating Reverb Effects at Initialization Time and Retrieving Instance Handles below.
- Pause (`sceScreamReverbPause()`) and continue (`sceScreamReverbContinue()`) reverb instances. See Pausing and Continuing Effect Instances below.
- Set stock presets (`sceScreamReverbSetStockPreset()`), and custom presets stored in `INI` files (`sceScreamReverbSetCustomPreset()`, `sceScreamReverbSetCustomPresetByName()`) on reverb instances. See Working with Stock I3DL2 Reverb Presets and Working with Custom Effect Presets below.
- Set all reverb instance properties explicitly (`sceScreamReverbSetAllProperties()`). See Setting All Effect Properties below.
- Set output level and panning (`sceScreamReverbSetVolumePolar()`) for reverb instances. See Setting Effect Instance Output Gain and Panning below.
- Set reverb instance output destination (`sceScreamReverbSetDirectPathOutputDest()`). See Setting Auxiliary Buss Output Destinations below.

For further details, see "Reverb Functions" in the *Scream Library Reference* documents.

Buss Configuration Functions

Note: The buss configuration functions are applicable to I3DL2 reverb instances created at initialization time only. To use these functions on the NGS2 synthesizer, do not set the `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API` initialization flag.

You use the buss configuration functions to set and query effect presets stored in an effect presets file (having a `BUS` extension) loaded at initialization time using the `SceScreamPlatformInitEx2` structure's *`pBussConfigFile`* member. With these functions you can:

- Apply a preset (`sceScreamApplyBussPreset()`) on a reverb instance.
- Retrieve the buss type associated with a named effect preset (`sceScreamGetBussPresetType()`).
- Retrieve a count of the number of effect presets stored in a loaded buss configuration file (`sceScreamGetBussPresetCount()`).
- Retrieve the name of an effect preset based on its index within a loaded buss configuration file (`sceScreamGetBussPresetName()`).

For further details, see:

- Working with BUS-Format Effect Presets Files in the "Working with Effect Presets" chapter.
- "Buss Configuration Functions" in the *Scream Library Reference* documents.

Effect Functions

Note: The effect functions are supported on the NGS2 synthesizer only, and are applicable to reverb, delay, and chorus effects instantiated during run time. To use these functions, you must set the `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API` initialization flag.

The effect functions allow you to manipulate reverb, delay, and chorus auxiliary effect instances. You can use the effect functions to:

- Create (`sceScreamEffectCreate()`) and destroy (`sceScreamEffectDestroy()`) effect instances. See Creating an Effect Instance and Inserting into an Auxiliary Buss below.
- Insert (`sceScreamEffectBussInsert()`) and remove (`sceScreamEffectBussRemove()`) effect instances into/from auxiliary busses. See Creating an Effect Instance and Inserting into an Auxiliary Buss below.
- Set auxiliary send gains and destinations on effect instances (`sceScreamEffectSetAuxSends()`).

- Pause (`sceScreamEffectPause()`) and continue (`sceScreamEffectContinue()`) effect instances. See Pausing and Continuing Effect Instances below.
- Set stock presets (`sceScreamEffectSetStockPreset()`), and custom presets stored in `INI` files (`sceScreamEffectSetCustomPreset()`, `sceScreamEffectSetCustomPresetByName()`) on effect instances. See Working with Stock I3DL2 Reverb Presets and Working with Custom Effect Presets below.
- Set all effect instance properties explicitly (`sceScreamEffectSetAllProperties()`). See Setting All Effect Properties below.
- Set effect instance output level and panning (`sceScreamEffectSetVolumePolar()`). See Setting Effect Instance Output Gain and Panning below.

For further details, see "Effect Functions" in the *Scream Library Reference* documents.

Effect Preset Functions

Note: The effect preset functions are supported on the NGS2 synthesizer only, and are applicable to reverb, delay, and chorus effects instantiated during run time. To use these functions, you must set the `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API` initialization flag.

You use the effect preset functions to set and query effect presets stored in an effect presets file (having a `BUS` extension) loaded at initialization time using the `SceScreamPlatformInitEx2` structure's *`pBussConfigFile`* member. With these functions you can:

- Apply a preset (`sceScreamApplyEffectPreset()`) on an effect instance.
- Retrieve the buss type or effect type associated with a named effect preset (`sceScreamGetEffectPresetType()`).
- Retrieve a count of the number of effect presets stored in a loaded buss configuration file (`sceScreamGetEffectPresetCount()`).
- Retrieve the name of an effect preset based on its index within a loaded buss configuration file (`sceScreamGetEffectPresetName()`).

For further details, see:

- Working with BUS-Format Effect Presets Files in the "Working with Effect Presets" chapter.
- "Effect Preset Functions" in the *Scream Library Reference* documents.

Auxiliary Buss Functions

You use the auxiliary buss functions to manipulate auxiliary busses. Currently, there is one function in this category (`sceScreamSetAuxBussOutputDest()`), which allows you to sets an auxiliary buss output destination. On the NGS synthesizer, the choice of output destinations includes pre-master submix busses as well as the master buss. On the NGS2 synthesizer, the choice of output destinations includes pre-master submix busses, the master buss, as well as the PlayStation®4 player-specific outputs.

For further details, see "Auxiliary Buss Functions" in the *Scream Library Reference* documents.

Understanding Auxiliary Buss Effects Signal Flow

Note: There is an important distinction between auxiliary buss effects and inline pre- and post-send filters. Inline filters are applied individually to Sounds. They affect the whole signal, and their outputs feed forward in series, one into the next. Auxiliary effects are usually applied collectively to Sounds. Their input signal is a branch of one or more Sounds routing through an auxiliary send. Their post-effect wet signal feeding in parallel back to the master buss (or into a pre-master submix buss).

Auxiliary buss effects require their input signal to be routed through an auxiliary buss from an auxiliary send. A Scream Sound has three auxiliary sends (`SCE_SCREAM_NUM_AUX_SENDS`), which can route Sound signal into any of the available auxiliary busses. The number of auxiliary busses is synthesizer-specific. On the NGS2 synthesizer, the number of auxiliary busses is configurable. You specify the number of auxiliary busses to instantiate at initialization time, using the `SceScreamSystemParams` structure's *`numAuxBusses`* member. On the NGS synthesizer there are three auxiliary busses (defined by the constant `SCE_SCREAM_NUM_AUX_BUSSES`).

Differences, with respect to synthesizer, in number of auxiliary busses, auxiliary sends, and routing capabilities are shown in the following table.

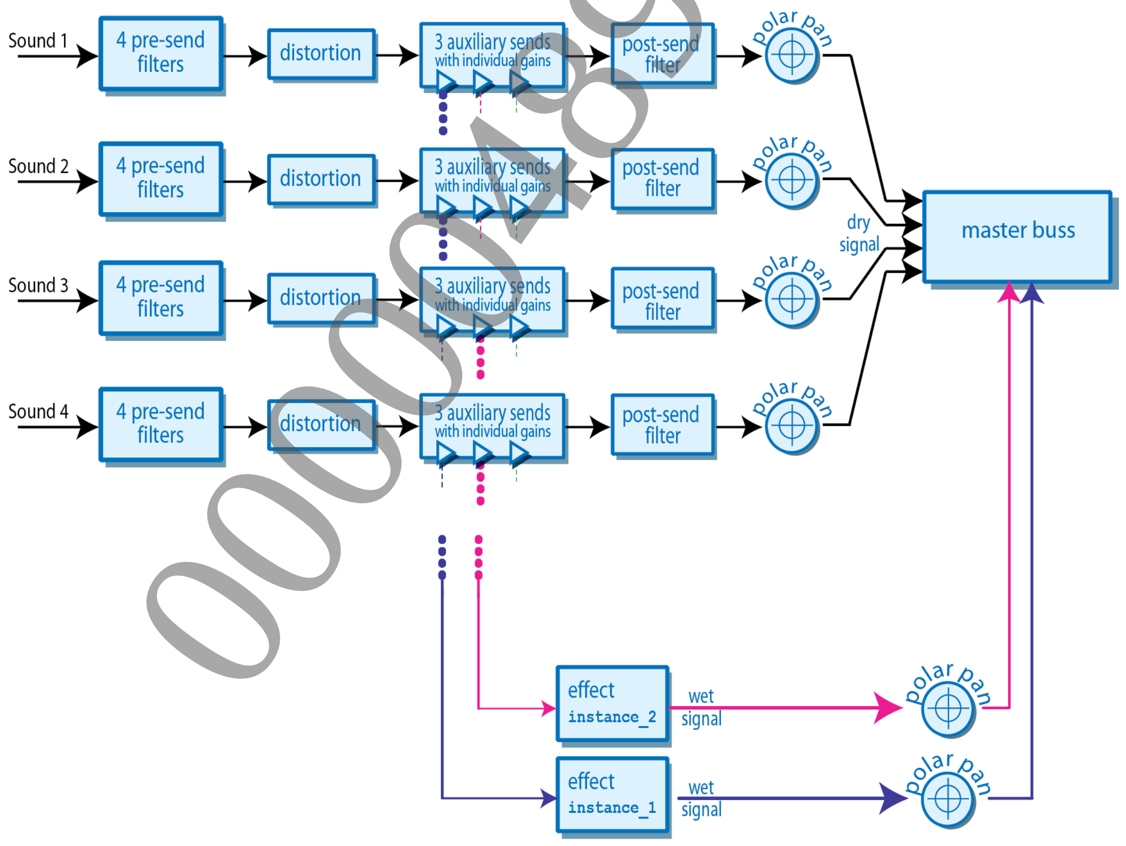
Table 26 Auxiliary Buss Routing Differences

Synthesizer	Auxiliary Sends	Auxiliary Busses	Routing
NGS	`SCE_SCREAM_NUM_AUX_SENDS` (3)	`SCE_SCREAM_NUM_AUX_BUSSES` (3)	Fixed
NGS2	`SCE_SCREAM_NUM_AUX_SENDS` (3)	`SceScreamSystemParams.`*`numAuxBusses`*	Assignable

Because auxiliary sends from multiple Sound instances can feed the same auxiliary buss, you can apply effects collectively to Sounds. This is ideal for environmental audio simulation: it allows Sounds that appear in the same environment to feed into the same effect. You control the amount of signal that feeds the auxiliary sends from each Sound with a gain, one for each send. See Configuring Sound Auxiliary Sends below.

To illustrate auxiliary buss effects signal flow, Figure 12 shows four Scream Sounds. Sounds 1 and 2 are feeding signal into auxiliary send 1, which is routed to auxiliary buss 1, where effect `instance_1` is attached. Sounds 3 and 4 are feeding signal into auxiliary send 2, which is routed to auxiliary buss 2, where effect `instance_2` is attached.

Figure 12 Auxiliary Buss Effects Signal Flow for Four Scream Sounds



You must configure auxiliary sends for each Sound from which you wish to route signal to an auxiliary buss effect.

Reverb effects instantiated at initialization time on the NGS or NGS2 synths are automatically inserted into available auxiliary busses. When instantiating auxiliary effects dynamically on the NGS2 synth, you must also insert effect instances into auxiliary busses.

To route signal from a Sound to an auxiliary effect, the following effect-specific and synth-specific tasks are required:

Table 27 Effect-specific and Synth-specific Signal Routing

Effect Type	NGS	NGS2
Reverb effects instantiated at initialization time	- Initialize Scream with the desired number of reverb instances, and retrieve their instance handles. See Instantiating Reverb Effects at Initialization Time and Retrieving Instance Handles below. - Configure Sound auxiliary send gains. See Configuring Sound Auxiliary Sends below.	- Initialize Scream with the desired number of reverb instances, and retrieve instance handles. See Instantiating Reverb Effects at Initialization Time and Retrieving Instance Handles below. - Configure Sound auxiliary send gains and destinations. See Configuring Sound Auxiliary Sends below.
Reverb, delay, and chorus effects dynamically instantiated during run time	Not supported.	- Dynamically create reverb, delay, and chorus effect instances and insert into auxiliary busses. See Creating an Effect Instance and Inserting into an Auxiliary Buss below. - Configure Sound auxiliary send gains and destinations. See Configuring Sound Auxiliary Sends below.

The Scream Reverb sample program may be useful as a source code reference.

Instantiating Reverb Effects at Initialization Time and Retrieving Instance Handles

You can instantiate I3DL2 reverb effects at initialization time by setting the `SceScreamSystemParams` structure's *`numReverbs`* member. Without a setting, the *`numReverbs`* member defaults to one reverb instance. However, you can set the *`numReverbs`* member per synthesizer to a maximum of `SCE_SCREAM_SND_MAX_REVERBS` reverb effects:

- NGS: 3 reverb effects (`SCE_SCREAM_SND_MAX_REVERBS` = 3)
- NGS2: 8 reverb effects (`SCE_SCREAM_SND_MAX_REVERBS` = 8)

Note: On the NGS2 synthesizer, you must *not* set the `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API` initialization flag, otherwise values for the `SceScreamSystemParams` *`numReverbs`* member are ignored.

Instantiated reverb voices are automatically inserted into the available auxiliary busses. You can determine the handle of a reverb instance assigned to a particular auxiliary buss using the `sceScreamReverbGetHandleByBuss()` function. You specify the buss to query in the function's *`buss`* parameter, using a zero-based index. The function returns a `SceScreamReverbHandle` identifying the reverb instance assigned to the queried auxiliary buss. You can use the returned `SceScreamReverbHandle` as input to other reverb functions. For further details, see Reverb Functions.

Creating an Effect Instance and Inserting into an Auxiliary Buss

Note: The procedures outlined in this section apply to the NGS2 synthesizer only.

To use the auxiliary buss delay and chorus effects on the NGS2 synthesizer, and to be able to dynamically instantiate these effects, plus the I3DL2 reverb effect, you must initialize Scream with the `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_EFFECT_API` flag set in the `SceScreamSystemParams` *`initFlags`* member. In addition, you must also instantiate effects, and insert them into an auxiliary busses.

The basic tasks involved are:

- Create an effect instance
- Assign a preset to the effect instance
- Insert the effect instance into an auxiliary buss

These tasks are described in the following sections.

Creating an Effect Instance

On the NGS2 synth you can dynamically create auxiliary reverb, delay, and chorus effect instances using the `sceScreamEffectCreate()` function. The maximum number of effect instances you can create is constrained by the value set at initialization time for the `SceScreamSystemParams` *`maxEffects`* member.

You specify the type of effect you wish to create in the `sceScreamEffectCreate()` function's *`type`* parameter, using one of the `SceScreamSndEffectTypes` enumerations. For further details, see `SceScreamSndEffectTypes` in the "Enumerations" chapter of the *Scream Library Reference* document for the PlayStation®4 platform.

The reverb, delay, and chorus effect types all require instantiation properties to be set when creating an instance. You specify these properties in the `sceScreamEffectCreate()` function's *`createProps`* parameter, pointing to an initialized `SceScreamSndReverbCreateProps`, `SceScreamSndDelayCreateProps`, or `SceScreamSndChorusCreateProps` structure, as appropriate to the type of effect being created.

Consultation Point: Designers may create effect presets that require effects to be instantiated with certain minimal resources. For example, regarding the delay effect, designers can set delay time and level for up to eight individual taps. The number of delay taps, however, is an instantiation property, set in the `SceScreamSndDelayCreateProps` structure's *`maxTaps`* member. Work with your design team to ensure that auxiliary effects are instantiated with appropriate resources called for in effect presets.

Note that you can also dynamically destroy effect instances using the `sceScreamEffectDestroy()` function.

Assigning a Preset to an Effect Instance

For details on assigning presets to an effect instance, see the following sections:

- Working with Stock I3DL2 Reverb Presets and Working with Custom Effect Presets later in this chapter.
- Working with BUS-Format Effect Presets Files and Working with INI-Format Effect Preset Files in the "Working with Effect Presets" chapter.

Insert an Effect Instance into an Auxiliary Buss

To insert an effect instance into an auxiliary buss you use the `sceScreamEffectBussInsert()` function. For the function's *`handle`* parameter, you specify the `SceScreamEffectHandle` of a particular effect instance to insert; which is a value returned by the `sceScreamEffectCreate()` function. And for the function's *`buss`* parameter, you specify a target auxiliary buss using a zero-based index.

Note that you can also remove an effect instance from an auxiliary buss into which it was inserted using the `sceScreamEffectBussRemove()` function.

Code Sample

The following code sample shows how to dynamically create an I3DL2 reverb instance, assign a stock preset to it, and insert the instance into an auxiliary buss:

```
// Initialize a SceScreamSndReverbCreateProps structure
SceScreamSndReverbCreateProps i3dl2Create;
i3dl2Create.reflectionsBufferSize = SCE_SCREAM_SND_I3DL2_REVERB_BUFFER_SIZE_SMALL;

// Create an I3DL2 reverb instance.
// The sceScreamEffectCreate() function returns a SceScreamEffectHandle.
g_i3dl2Reverb = sceScreamEffectCreate(SCE_SCREAM_SND_EFFECT_TYPE_REVERB, &i3dl2Create);

// Assign the stock I3DL2 preset "Cave" to the reverb instance.
sceScreamEffectSetStockPreset(g_i3dl2Reverb, SCE_SCREAM_SND_I3DL2_ENVIRONMENT_PRESET_CAVE);

// Insert the instantiated I3DL2 reverb effect into auxiliary buss 0
sceScreamEffectBussInsert(g_i3dl2Reverb, 0);
```

Having performed these tasks, if auxiliary sends from one or more Sounds are routed to auxiliary buss 0, they feed into the instantiated I3DL2 reverb instance attached to that buss, upon which the "Cave" preset is assigned.

Configuring Sound Auxiliary Sends

The NGS2 synthesizer has three assignable auxiliary sends, routing signal to a choice of up to eight auxiliary busses, determined at initialization time by the value set for the `SceScreamSystemParams` *`numAuxBusses`* member. The NGS synthesizer has three fixed auxiliary sends, routing signal to three corresponding auxiliary busses. The procedure for configuring Sound auxiliary sends is therefore synthesizer-specific.

To configure Sound auxiliary send gains and destinations on the NGS2 synthesizer:

(1) Set gain levels for the auxiliary sends you wish to route signal from using an array of gain values applied to the `SceScreamSynthParams` *`auxSendGain`* member. Values are in the range 0 to 1.0.

(2) Set the destination for each auxiliary send using an array of zero-based auxiliary buss indices applied to the `SceScreamSynthParams` *`auxSendDests`* member.

To configure Sound auxiliary sends on the NGS synthesizer:

(1) Set gain levels for the auxiliary sends corresponding with target auxiliary busses you wish to route signal to using an array of gain values applied to the `SceScreamSynthParams` *`auxSendGain`* member. Values are in the range 0 to 1.0.

(2) Auxiliary sends have fixed destinations on NGS: auxiliary send 0 → auxiliary buss 0; auxiliary send 1 → auxiliary buss 1; and auxiliary send 2 → auxiliary buss 2. Therefore, there is no need to set the auxiliary send destination to the number of the buss to which a target effect instance is attached. Settings to the `SceScreamSynthParams` *`auxSendDests`* member are ignored.

Note: Both the `SceScreamSynthParams` *`auxSendGain`* and *`auxSendDests`* members take an array of three values representing each of the auxiliary sends. Values specified in these arrays are not persistent between calls that set `SceScreamSynthParams` (by setting `SceScreamSoundParams`). For example, if you set *`auxSendGain`* in one call to `[1.0, 1.0, 1.0]`, and later, in another call, set it to `[.5, .5]`, the value for the 3rd auxiliary send gain missing from the 2nd call will revert to `0.0`, not persist at `1.0`.

The following code shows how to configure Sound auxiliary send gains, and assumes that an effect instance is assigned to auxiliary buss 0.

```
// initialize a SceScreamSoundParams structure, and set the mask member to
// specify that only the embedded SceScreamSynthParams structure is being set
SceScreamSoundParams my_soundparams;
memset(&my_soundparams, 0, sizeof(SceScreamSoundParams));

my_soundparams.size = sizeof(SceScreamSoundParams);
my_soundparams.mask = SCE_SCREAM_SND_MASK_SYNTH_PARAMS;
```

```
    my_soundparams.synthParams.mask = SCE_SCREAM_SND_MASK_AUXSEND_GAINS;

    // set a gain value for auxiliary send 0
    my_soundparams.synthParams.auxSendGain[0] = 0.75;

    // Note: on the NGS synth the destination is fixed at auxiliary buss 0.
    // Therefore the following line applies to NGS2 only.
    my_soundparams.synthParams.auxSendDests[0] = 0;

    // apply the above settings to the Sound with handle mySoundHandle
    sceScreamSetSoundParamsEx(mySoundHandle, &my_soundparams);
```

Working with Stock I3DL2 Reverb Presets

The I3DL2 reverb's 30 stock presets provide a range of settings to simulate the acoustics of diverse environments. Audio designers can select the presets in Scream Tool, and either use them as is, or customize them to suit the specifics of a particular environment. For a list of I3DL2 presets see the `SceScreamI3DL2StockPresets` enumeration in the *Scream Library Reference* documents. Because stock presets are pre-loaded, no presets file is required.

On the NGS synth or NGS2 synth using legacy effects behavior, you set a stock preset on a reverb instance using the `sceScreamReverbSetStockPreset()` function. The function takes as input a reverb instance handle (`SceScreamReverbHandle`; obtainable by calling the `sceScreamReverbGetHandleByBuss()` function), and a preset index.

On the NGS2 synth, with effect API enabled, you set a stock preset on a reverb instance using the `sceScreamEffectSetStockPreset()` function. The function takes as input an effect instance handle (`SceScreamEffectHandle`; returned by the `sceScreamEffectCreate()` function), and a preset index.

Note: The I3DL2 reverb default preset (`SCE_SCREAM_SND_I3DL2_ENVIRONMENT_PRESET_DEFAULT`) sets the *Room* parameter to -10,000 mB. In other words, reverb output level is turned all the way down, and is therefore inaudible, equivalent to being off.

Note: There are no stock presets for the delay and chorus effects.

Working with Custom Effect Presets

Audio designers can audition and save auxiliary effect presets in Scream Tool using the Buss Editor. Effect presets however, are not included in Bank contents, and custom presets must be exported separately from Scream Tool (using a binary file format with a `BUS` extension), for integration into the Scream Runtime. Using the `BUS` presets file format, you can set custom presets on effect instances using either:

- NGS and NGS2 legacy effects behavior: the `sceScreamApplyBussPreset()` function.
- NGS2 effect API enabled: the `sceScreamApplyEffectPreset()` function.

See Working with BUS-Format Effect Presets Files and Applying an Effect Preset in the "Working with Effect Presets" chapter for details.

Designers can also export I3DL2 reverb presets in the `INI` file format; a text list format, easily loaded into the Scream Runtime. For further details see Working with INI-Format Effect Preset Files in the "Working with Effect Presets" chapter. Using the `INI` presets file format, you can set a custom preset on a reverb instance, either by reference to its name or index within a loaded presets `INI` file, using either:

- NGS and NGS2 legacy effects behavior: the `sceScreamReverbSetCustomPresetByName()` or `sceScreamReverbSetCustomPreset()` functions.
- NGS2 effect API enabled: the `sceScreamEffectSetCustomPresetByName()` or `sceScreamEffectSetCustomPreset()` functions.

Setting All Effect Properties

You can explicitly set all properties on reverb or other auxiliary buss effects using either:

- NGS and NGS2 legacy effects behavior: the `sceScreamReverbSetAllProperties()` function.
- NGS2 effect API enabled: the `sceScreamEffectSetAllProperties()` function.

Setting all effect properties directly, rather than setting an effect preset, can be a more complex approach. However, it may be useful if you are working with a custom data format for storing effect settings or programmatically generating effect property values based on game geometries.

The `sceScreamReverbSetAllProperties()` or `sceScreamEffectSetAllProperties()` functions' *`properties`* parameter points to an effect-specific structure storing effect property values as shown in the following table:

Table 28 Effect-specific Property Structures

Effect Type	Properties Structure
Reverb	`SceScreamSndReverbProps`
Delay	`SceScreamSndDelayProps`
Chorus	`SceScreamSndChorusProps`

Note: Setting all effect or all reverb properties overrides all current properties set on an effect instance arising from an applied preset. You cannot use the `sceScreamReverbSetAllProperties()` or `sceScreamEffectSetAllProperties()` functions to set only a subset of effect properties, leaving the remaining properties from an applied preset intact. The only way to accomplish partial setting of effect properties on top of an applied preset would be to first store the preset's property values in the appropriate effect data structure, modify a desired subset of the properties, and then apply all properties (including the modified values) from the structure.

I3DL2 Reverb Parameters

The I3DL2 reverb offers a number of parameters, allowing you to specify settings to simulate a wide variety of environments – from closet-size enclosures to forests and quarries. Table 29 shows the I3DL2 reverb parameters as exposed in the `SceScreamSndReverbProps` structure in the Scream API.

Table 29 Exposed I3DL2 Reverb Parameters

Parameters	Value	Description
`Room`	Range: -10000.0 to 0.0 mB	Output level of wet (reverb-treated) signal, in mB (milliBells; 100 mB = 1 dB).
`Room_HF`	Range: -10000.0 to 0.0 mB; where 0.0 produces no coloration	Attenuation of high-frequencies, in mB, with respect to the *`HF_reference`* value.
`Decay_time`	Range: 0.1 to 20 seconds	Late reverberation (diffuse tail) decay time. Larger, more reflective environments have longer decay times. Smaller, less reflective environments have shorter decay times.
`Decay_HF_ratio`	Range: 0.1 to 2.0	Ratio of late reverberation high-frequency decay to low-frequency decay with respect to the *`HF_reference`* value.
`Reflections`	Range: -10000 to +1000 mB	Early reflections level, in mB. Use this parameter in conjunction with the *`Reverb`* value to set the balance between early reflections and late reverberation.

Parameters	Value	Description
Reflections_delay	Range: 0.0 to 0.3 seconds	Pre-delay time before the onset of early reflections. Simulates room size. Larger spaces have a longer pre-delay before the onset of early reflections.
Reverb	Range: -10000 to +2000 mB	Late reverberation level, in mB. Use this parameter in conjunction with the *Reflections* value to set the balance between early reflections and late reverberation.
Reverb_delay	Range: 0.0 to 0.1 seconds	Delay time before the onset of late reverberation. Simulates room size; larger spaces have a longer delay before the onset of late reverberation.
Diffusion	Range: 0 to 100%	Echo density of late reverberation. Simulates the reflectivity of the environment's surfaces.
Density	Range: 0 to 100%	Modal density of late reverberation.
HF_reference	Range: 20 to 20000 Hertz	Reference high frequency. Used in conjunction with *Room_HF* and *Decay_HF_ratio*.
EarlyReflectionPattern	Range: …_ROOM1_LEFT to …_ROOM3_RIGHT	One of the Early Reflections Pattern Constants per channel.
EarlyReflectionScalar	Range: 0 to 100%; where 0% specifies single reflection, and 100% specifies widely spread reflections	Scales the early reflections spread. Higher values produce more spread-out early reflections (as found in larger spaces); lower values produce more tightly-packed early reflections (as found in smaller spaces).
LF_reference	Range: 20 to 20000 Hertz	Reference low frequency. Used in conjunction with *Room_LF*.
Room_LF	Range: -10000 to 0 mB	Attenuation of low-frequencies with respect to the *LF_reference* value.
DryMB	Range: -10000 to 0 mB	Output level of dry (untreated) signal, in mB.

Setting Effect Instance Output Gain and Panning

To hear an auxiliary effect you must set the instance's output gain and panning. You do this by calling either of the following functions with appropriate arguments for gain, azimuth, and focus:

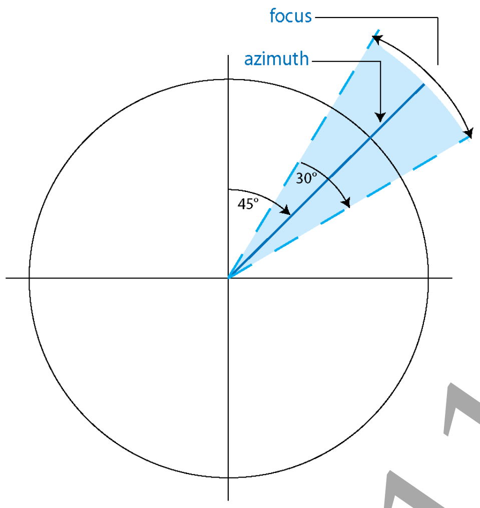
- NGS and NGS2 legacy effects behavior: `sceScreamReverbSetVolumePolar()`
- NGS2 effect API enabled: `sceScreamEffectSetVolumePolar()`

Notes:
- When running Scream on the NGS synthesizer the following parameters of the `sceScreamReverbSetVolumePolar()` functions are ignored: *lfeGain*, *mode*, *excludeTargets*.
- When running Scream on the NGS2 synthesizer the following parameters of the `sceScreamReverbSetVolumePolar()` function are ignored: *mode*, *excludeTargets*.

Azimuth and Focus

Azimuth and focus are two controls used to specify the spatial placement of wet (reverberated) Sounds into the panning field. Azimuth specifies the direction of a Sound, and focus specifies its width. If focus is set to 360 degrees, azimuth becomes irrelevant, as reverb is equally projected from all directions. Figure 13 depicts spatial placement of a Sound with an azimuth of 45 degrees and a focus of 30 degrees.

Figure 13 Azimuth and Focus



Setting Auxiliary Buss Output Destinations

Using the `sceScreamSetAuxBussOutputDest()` function, you can route auxiliary buss output to a choice of destinations:

- To the master output (the default)
- To a pre-master submix buss
- On PlayStation®4 only, to a custom player-specific voice, pad speaker, or stereo output

Routing to master buss output is the default behavior if you do not call the `sceScreamSetAuxBussOutputDest()` function. However, if you have routed auxiliary buss output to a destination other than the master buss, and wish to re-route output back to the master buss, you can set the `sceScreamSetAuxBussOutputDest()` function's *`outputDest`* parameter to `SCE_SCREAM_SND_OUTPUT_DEST_MASTER`.

To route auxiliary buss output to a pre-master submix buss, you set the *`outputDest`* parameter to the zero-based index of the desired submix buss. See Setting NGS/NGS2 Pre-Master Submix Indices in the "Working with Pre-Master and Master Signal Processors" chapter for details.

Note: Pre-master submix busses must be allocated at initialization time. You allocate pre-master submix busses using the *`numPremasterCompSubmixes`* and *`numPremasterScCompSubmixes`* members of the `SceScreamSystemParams` structure. See Allocating Pre-Master Submixes in the "Configuration, Initialization, and Shutdown" chapter for further details.

The index of the first pre-master submix is expressed in the constant `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0`, equal to zero. Permissible pre-master submix indices range from zero to `(SCE_SCREAM_SND_MAX_PREMASTER_SUBMIXES - 1)`, that is 0 to 3.

Routing to Custom Personal Outputs on PlayStation®4

To route auxiliary buss output to a custom player-specific output on PlayStation®4, you set the *`outputDest`* parameter to a combination of a local player identifier and a personal output destination, encapsulated within the `SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM` macro. For example, to route auxiliary buss output to the pad speaker output for player 0, you set the *`outputDest`* parameter as follows:

```
SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM(SCE_SCREAM_SND_OUTPUT_DEST_PLAYER_0,
SCE_SCREAM_SND_OUTPUT_DEST_PERSONAL_PAD_SPEAKER)
```

To use custom personal output destinations, you must initialize Scream's record of logged-in local players using members of the `SceScreamSystemParams` structure:

- *`maxLocalPlayers`*
  Sets the maximum number of local players that can be logged-in at any one time. Range: 1 to 4.
- *`numLocalPlayers`*
  Sets the number of local players that are currently logged-in. Range: 1 to *`maxLocalPlayers`*.
- *`localPlayerIDs`*
  An array of user IDs of currently logged-in local players.

If the number of players or specific player IDs change during game play, you can use the `sceScreamSynthUpdateLocalPlayerIDs()` function to dynamically update Scream's records.

You must also enable any custom output ports you intend to use by setting the appropriate NGS2 initialization flag in the `SceScreamSystemParams` structure's *`initFlags`* member corresponding to the personal output destination:

- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PAD_SPEAKER`
  Opens pad speaker output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PERSONAL_STEREO`
  Opens personal stereo (2-channel) ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_VOIP`
  Opens VoIP headset output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_BGM`
  Opens the background music (BGM) port.

For further details, see the *Scream Library Reference* document for the PlayStation®4 platform:

- "Sound Output Destinations" in the "Constants" chapter
- `SceScreamSystemParams` in the "NGS2 Data Structures" chapter

See also:

- Configuring PlayStation®4 Custom Personal Outputs
- Setting Sound Output Destinations
- Setting Group Output Destinations
- Routing Pre-Master Submix Output Destination

Pausing and Continuing Effect Instances

You can pause and continue effect or reverb instances using:

- NGS and NGS2 legacy effects behavior: the `sceScreamReverbPause()` and `sceScreamReverbContinue()` functions.
- NGS2 effect API enabled: the `sceScreamEffectPause()` and `sceScreamEffectContinue()` functions.

These functions allow you to silence effect output simultaneously when pausing Sounds using either Group or individual Sound pause functions. Otherwise, in the case of reverb effects, if output is not paused, and depending on the decay time, the reverb tail may continue noticeably after a Sound is paused.

On the NGS and NGS2 synthesizers, the `sceScreamEffectPause()` and `sceScreamReverbPause()` functions suspend processing of I3DL2 instances. This means that active reverb tails are preserved, and are resumed immediately upon continuing the instance.

Typically the pause and continue functions are used in conjunction with Sound (or Group) pause functions, as in the following example:

```
sceScreamPauseSound(mySoundHandle);
sceScreamEffectPause(myEffectHandle);
...
sceScreamContinueSound(mySoundHandle);
sceScreamEffectContinue(myEffectHandle);
```

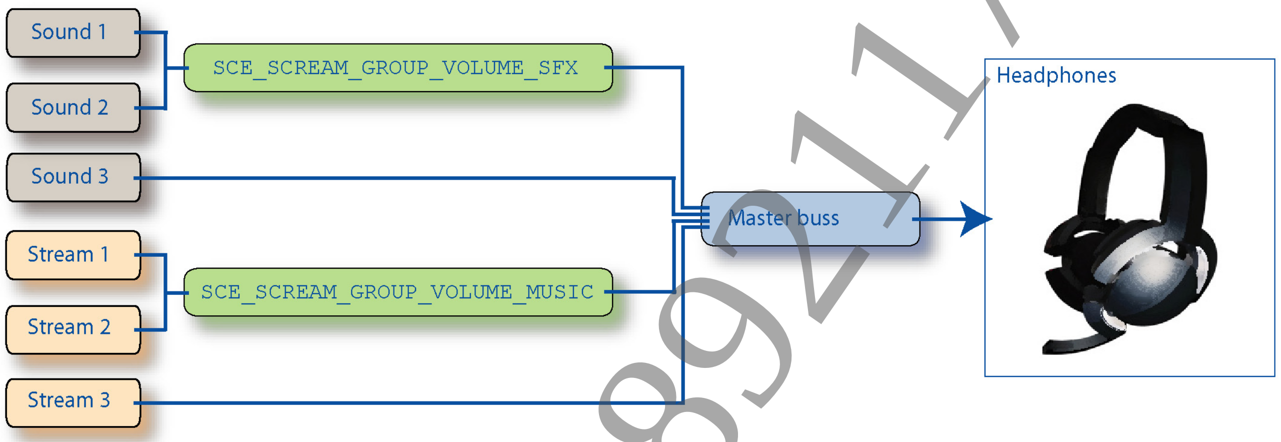
10 Working with Pre-Master and Master Signal Processors

The NGS and NGS2 synthesizers offer signal processing units on the pre-master submix and master busses. This chapter guides you through tasks related to configuring and using these processing units.

Pre-Master Submix and Master Buss Signal Flow

By default, voice output from Sounds, Streams, and Volume Groups routes directly to the master buss. An example configuration is shown in Figure 14.

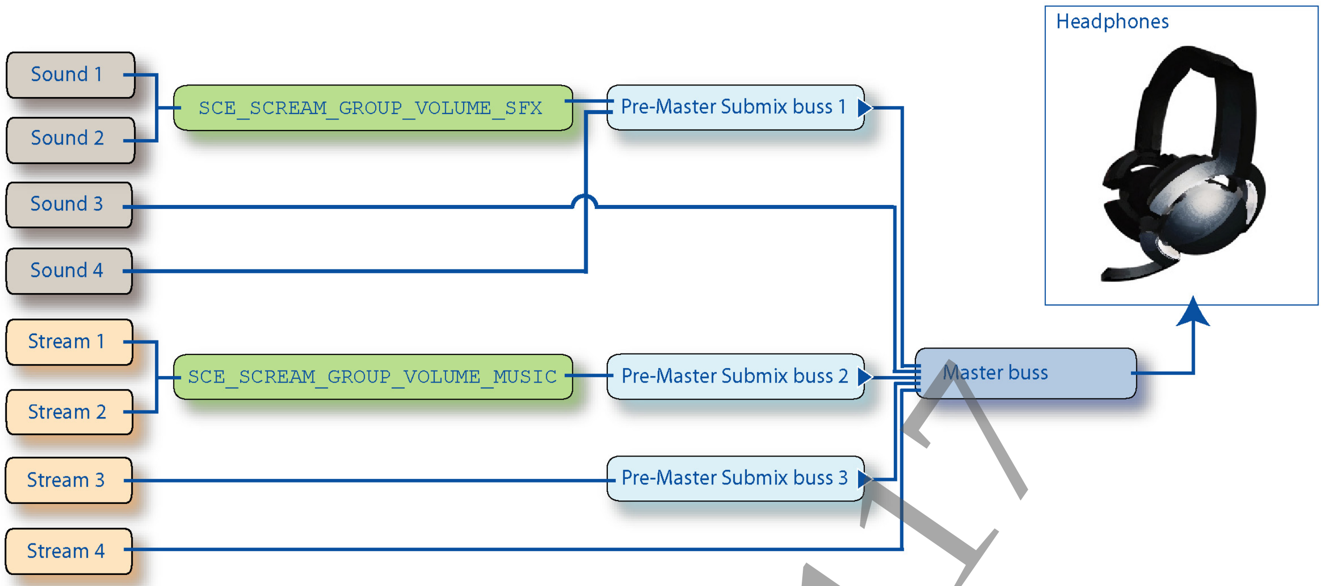
Figure 14 Default Voice Signal Routing to the Master Buss



Alternatively, you can route voice output from Sounds and Groups to a pre-master submix buss. Routing to a pre-master submix enables pre-mixing and treatment of combined signal from Sound and Group voices prior to it entering the master buss. For details about setting Sound and Group output destinations, see Setting Sound Output Destinations in the "Working with Sounds" chapter and Setting Group Output Destinations in the "Working with Groups" chapter.

Figure 15 shows the same Sound, Stream, and Group voice resources as in Figure 14. In this custom configuration, signal from the SFX and Music Groups routes to the master buss through pre-master submix busses 1 and 2, signal from Sound 3 retains its default routing directly to the master buss, and signal from Stream 3 routes to the master buss through pre-master submix buss 3.

Figure 15 Custom Voice Signal Routing to Pre-Master Submix, and Master Busses



Working with Custom Presets

Audio designers can audition and save NGS and NGS2 pre-master submix buss effects settings in Scream Tool using the Buss Editor. Buss effects settings however, are not included in Bank data, and any custom presets must be exported separately from Scream Tool, using either the `BUS` (binary) or `INI` (text) file formats.

If using the `BUS` presets file format, you can set a custom preset on a pre-master submix buss, or the master buss by using the `sceScreamApplyBussPreset()` function. See Working with BUS-Format Effect Presets Files and Applying an Effect Preset in the "Working with Effect Presets" chapter for details.

Alternatively, it is possible to manually create presets files in the `INI` file format; a text list format, easily loaded into the Scream runtime. See Working with INI-Format Effect Preset Files in the "Working with Effect Presets" chapter for details. If using the `INI` presets file format, you can set a custom preset on a pre-master submix buss or the master buss, either by reference to its name or index within a loaded presets (`INI`) file, using the `sceScreamPremasterSubmixSetCustomPresetByName()` or `sceScreamPremasterSubmixSetCustomPreset()` functions. For further details, see the *Scream Library Reference* documents.

Setting All Buss Effects Properties Programmatically

You can collectively set the properties of all pre-master submix buss effects using the `sceScreamPremasterSubmixSetAllProperties()` function. You specify the index of the submix buss upon which to set all properties in the function's *`premasterSubmixID`* parameter. See Setting NGS/NGS2 Pre-Master Submix Indices below for details. You can store pre-master submix buss effect property values using a `SceScreamSndPremasterSubmixProps` structure. Point to an instance of this structure, appropriately initialized with property values for all pre-master submix buss effects, in the `sceScreamPremasterSubmixSetAllProperties()` function's *`properties`* parameter.

Routing Pre-Master Submix Output Destination

Note: Routing pre-master submix output is applicable to the NGS2 synthesizer only.

On the NGS2 synthesizer you can set a pre-master submix output destination using the `sceScreamSynthPremasterSubmixSetOutputDest()` function. The function connects the output of a pre-master submix to any of the following output destinations:

- the input of another pre-master submix
- the master output buss
- a custom player-specific voice, pad speaker, or stereo output

Linking together pre-master submix busses in series allows you to leverage additional signal processing capabilities. In the `sceScreamSynthPremasterSubmixSetOutputDest()` function's two parameters, you specify the index of a pre-master submix to set, and an output destination (the index of another pre-master submix or the master output buss). See Setting NGS/NGS2 Pre-Master Submix Indices below for further details.

Note: Avoid feedback loops. Do not route a pre-master submix output to its own input, directly or indirectly. For instance, do not route submix 0 output to submix 1 input and then route submix 1 ouput to submix 0 input.

Without calling the `sceScreamSynthPremasterSubmixSetOutputDest()` function, the pre-master submix output destination defaults to the master buss. If you have routed pre-master submix output to another pre-master submix, you can use the `sceScreamSynthPremasterSubmixSetOutputDest()` function to re-route output to the master buss. For further details, see the `sceScreamSynthPremasterSubmixSetOutputDest()` function in the *Scream Library Reference* document for the PlayStation®4 platform.

Routing to Custom Personal Outputs on PlayStation®4

To route pre-master submix signal to a custom player-specific output on PlayStation®4, you set the `sceScreamSynthPremasterSubmixSetOutputDest()` function's *`outputDest`* parameter to a combination of a local player identifier and a personal output destination, encapsulated within the `SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM` macro. For example, to route to the pad speaker output for player 0, you set the *`outputDest`* parameter as follows:

```
SCE_SCREAM_SND_OUTPUT_DEST_CUSTOM(SCE_SCREAM_SND_OUTPUT_DEST_PLAYER_0,
SCE_SCREAM_SND_OUTPUT_DEST_PERSONAL_PAD_SPEAKER)
```

To use custom personal output destinations, you must initialize Scream's record of logged-in local players using members of the Sce`ScreamSystemParams` structure:

- *`maxLocalPlayers`*
  Sets the maximum number of local players that can be logged-in at any one time. Range: 1 to 4.
- *`numLocalPlayers`*
  Sets the number of local players that are currently logged-in. Range: 1 to *`maxLocalPlayers`*.
- *`localPlayerIDs`*
  An array of user IDs of currently logged-in local players.

If the number of players or specific player IDs change during game play, you can use the `sceScreamSynthUpdateLocalPlayerIDs()` function to dynamically update Scream's records.

You must also enable any custom output ports you intend to use by setting the appropriate NGS2 initialization flag in the `SceScreamSystemParams` structure's *`initFlags`* member corresponding to the personal output destination:

- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PAD_SPEAKER`
  Opens pad speaker output ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_PERSONAL_STEREO`
  Opens personal stereo (2-channel) ports for local players.
- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_VOIP`
  Opens VoIP headset output ports for local players.

- `SCE_SCREAM_SND_SYNTH_INIT_FLAG_ENABLE_BGM`
  Opens the background music (BGM) port.

For further details, see the *Scream Library Reference* document for the PlayStation®4 platform:

- "Sound Output Destinations" in the "Constants" chapter
- `SceScreamSystemParams` in the "NGS2 Data Structures" chapter

See also:

- Configuring PlayStation®4 Custom Personal Outputs
- Setting Group Output Destinations
- Setting Sound Output Destinations
- Setting Auxiliary Buss Output Destinations

Setting NGS/NGS2 Pre-Master Submix Indices

The first parameter in the pre-master submix buss functions is *`premasterSubmixID`*. This is the zero-based index of a pre-master submix upon which to operate. The range of index values you can specify depends on the number of pre-master submix busses allocated at initialization time. On the NGS and NGS2 synths there are two types of pre-master submix voice:

- submix with standard (inline) compression
- submix with side-chain compression

You allocate these pre-master submix voices using the *`numPremasterCompSubmixes`* and *`numPremasterScCompSubmixes`* members of the `SceScreamSystemParams` structure. See Allocating Pre-Master Submixes in the "Configuration, Initialization, and Shutdown" chapter for further details.

Pre-master submix indices begin with `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0` (equal to zero) for the first submix with standard compression, ascending through any additional submixes of that type. Indices for pre-master submixes with side-chain compression continue from the last index for a submix with standard compression. If no pre-master submixes with standard compression are allocated, the `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0` constant represents the first submix with side-chain compression.

For example, suppose you have allocated two pre-master submixes with standard compression, and two submixes with side-chain compression. Indices for these submixes are as follows:

- 1st submix with standard compression: `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0`
- 2nd submix with standard compression: `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0` + 1
- 1st submix with side-chain compression: `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0` + 2
- 2nd submix with side-chain compression: `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0` + 3

If you had allocated zero pre-master submixes with standard compression, and two submixes with side-chain compression, indices for these submixes are as follows:

- 1st submix with side-chain compression: `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0`
- 2nd submix with side-chain compression: `SCE_SCREAM_SND_OUTPUT_DEST_PREMASTER_0` + 1

Pre-Master Submix Voices

Pre-master submix voices allow you add special pre-master treatment and to pre-mix Groups or individual Sounds. By setting the `SceScreamSystemParams` structure's *`numPremasterCompSubmixes`* and *`numPremasterScCompSubmixes`* members, you can allocate up to four (`SCE_SCREAM_SND_MAX_PREMASTER_SUBMIXES`) pre-master submix voices of each type when initializing Scream. See Allocating Pre-Master Submixes for further details.

You route signal from a Group or Sound to a pre-master submix voice by setting the `outputDest` parameter of the `sceScreamSetGroupVoiceOutputDest()`, `sceScreamPlaySoundByIndexEx()`, or `sceScreamPlaySoundByNameEx()` functions. See Setting Group Output Destinations in the "Working with Groups" chapter and Setting Sound Output Destinations in the "Working with Sounds" chapter for further details.

Pre-master submix busses on NGS and NGS2 provide the following signal processing units:

- **Compressor** – Dynamic range compression. Supports optional side-chain input from another pre-master submix voice.
- **Output Gain** – Sets send level to the master buss.

Compressor

Pre-master submix busses provide a compressor, enabling dynamic range compression of submix signal prior to it entering the master buss. Whether a compressor supports side-chain input from another pre-master submix voice is determined at initialization time. You specify the number of pre-master submix voices to allocate without side-chain input in the *`numPremasterCompSubmixes`* member of the `SceScreamSystemParams` structure. Specify the number of pre-master submix voices to allocate with side-chain input in the *`numPremasterScCompSubmixes`* member. With side-chain compressor input, signal analysis is performed on the side-chain signal rather than the main input signal. This acts as a kind of automated ducker, and can be useful in voice over contexts. To connect side-chain input from the output of one pre-master submix to the compressor input on another submix, you use the `sceScreamSynthPremasterSubmixConnectSideChainInput()` function.

You set up a pre-master submix compressor using the `sceScreamSynthPremasterSubmixSetupCompressor()` function. You can specify optional compression features, such as channel linking, and the choice of peak or RMS compression mode. Channel linking preserves the original panning image, and is specified in the *`linkedChannels`* parameter using a Boolean value. Note, however, that channel linking is a more intrusive compression mode, because each channel is compressed equally based on the `peakMode` setting. You specify the choice of peak or RMS mode in the `peakMode` parameter using a Boolean value. In peak mode (`TRUE`), the compressor responds to the instantaneous level of the input signal. Peak mode can produce more quick-reacting and obvious results. In RMS mode (`FALSE`, the default), the compressor responds to an averaged level of the input signal. RMS mode can produce more relaxed and subtle results. You can specify values for compression parameters such as threshold, ratio, attack time, release time, make-up gain (dB), and knee curve in the remaining parameters. For further details, see the `sceScreamSynthPremasterSubmixSetupCompressor()` function in the *Scream Library Reference* documents.

Output Gain

You set the pre-master submix output level to the master buss by using the `sceScreamSynthPremasterSubmixSetOutputGain()` function. See the *Scream Library Reference* documents for further details.

Master Voice

The master voice allows you to prepare pre-mixed audio (routed from pre-master submix busses, Groups, and individual Sounds) for final output. The master buss on NGS and NGS2 provides the following signal processing units:

- **Compressor** – Dynamic range compression.

Compressor

The master buss provides a compressor, enabling dynamic range compression of master buss signal.

You set up the master buss compressor using the `sceScreamSynthMasterSetupCompressor()` function, which has very similar parameters to `sceScreamSynthPremasterSubmixSetupCompressor()`. You can specify optional compression features such as channel linking, and the choice of peak or RMS compression mode. Channel linking preserves the original panning image, and is specified in the `linkedChannels` parameter using a Boolean value. Note however, that channel linking is a more intrusive compression mode because each channel is compressed equally based on the peakMode setting. You specify the choice of peak or RMS mode in the `peakMode` parameter using a Boolean value. In peak mode (`TRUE`) the compressor responds to the instantaneous level of the input signal. Peak mode can produce more quick-reacting and obvious results. In RMS mode (`FALSE`, the default) the compressor responds to an averaged level of the input signal. RMS mode can produce more relaxed and subtle results. You can specify values for compression parameters such as threshold, ratio, attack time, release time, make-up gain (dB), and knee curve in the remaining parameters. For further details, see the `sceScreamSynthMasterSetupCompressor()` function in the *Scream Library Reference* documents.

11 Working with Effect Presets

Scream supports two alternative file formats for buss presets: a binary effect preset file (having a `BUS` extension), and a text list file (having an `INI` extension). Designers can export `BUS` effect preset files from Scream Tool, containing presets for auxiliary, pre-master, and master buss effects on the NGS and NGS2 synthesizers. Audio designers can also export `INI` files from Scream Tool, but only for I3DL2 reverb presets. `INI` files containing presets for pre-master submix and master buss effects must be created manually.

Working with BUS-Format Effect Presets Files

Designers can create buss effects presets in Scream Tool, and export them to the binary effect preset file format (having a `BUS` extension). Effect preset files contain one or more presets. Each preset defines parameter settings for one buss effect on the NGS or NGS2 synthesizers. That is, each preset can contain settings for one of the following only:

- Reverb effect on a NGS2 auxiliary buss
- Delay effect on a NGS2 auxiliary buss
- Chorus effect on a NGS2 auxiliary buss
- Reverb effect on a NGS auxiliary buss
- Pre-master submix buss effects on the NGS2 synth
- Pre-master submix buss effects on the NGS synth
- Master buss effects on the NGS2 synth
- Master buss effects on the NGS synth

Loading an Effect Preset File

To load a `BUS`-format effect preset file into the Scream Runtime, you must first pre-load the file into memory. You then provide the corresponding memory pointer as a value for the `SceScreamPlatformInitEx2` structure's *`pBussConfigFile`* member when initializing Scream.

Note: While still present in the `SceScreamPlatformInitEx2` structure, the *`bussConfigFileSize`* member is ignored.

Buss Configuration Functions versus Effect Preset Functions

On the NGS synthesizer, you use the buss configuration functions to set and query buss effect presets.

On the NGS2 synthesizer, you either use the buss configuration functions or the effect preset functions, depending on whether you choose to initialize Scream with legacy effects behavior or with effect API enabled:

- legacy effects behavior: use the buss configuration functions.
- effect API enabled: use the effect preset functions.

The functionality of these two sets of functions is identical. The main difference is that the effect preset function, `sceScreamApplyEffectPreset()`, requires a dynamically-created `SceScreamEffectHandle` as input, whereas the equivalent `sceScreamApplyBussPreset()` function does not.

For further details, see and Buss Configuration Functions and Effect Preset Functions in the "Working with Auxiliary Buss Effects" chapter.

Applying an Effect Preset

Having loaded a `BUS`-format effect preset file, you can apply the presets it contains using either:

- NGS and NGS2 legacy effects behavior: the `sceScreamApplyBussPreset()` function.
- NGS2 effect API enabled: the `sceScreamApplyEffectPreset()` function.

You specify the name of a preset to apply in the function's *`name`* parameter. You can optionally specify the index of a buss upon which to apply the preset in the function's *`bussIndex`* parameter. Note that the buss index can also be included with preset data, in which case specifying a buss index with the function call overrides any preset-specified buss index. Buss indices are only valid where there are multiple busses of the same type. That is, buss indices are valid for auxiliary and pre-master submix buss presets, but not for master buss presets. To ensure that the buss index value is specified by the preset, you can set the *`bussIndex`* parameter to -1.

Querying Effect Presets

You can query the presets contained in a loaded effect preset file using functions shown in the following table. On the NGS synth, you use the buss configuration functions, shown in the left column. On the NGS2 synth, depending on whether you initialize Scream with legacy effects behavior or with effect APIs enabled, you use either the functions in the left or middle columns.

Table 30 Effect Preset Query Functions

NGS and NGS2 Legacy Behavior	NGS2 Effect API Enabled	Description
`sceScreamGetBussPresetType()`	`sceScreamGetEffectPresetType()`	Retrieves the buss type associated with a named preset.
`sceScreamGetBussPresetCount()`	`sceScreamGetEffectPresetCount()`	Retrieves the total count of effect presets contained in an effect presets file.
`sceScreamGetBussPresetName()`	`sceScreamGetEffectPresetName()`	Retrieves the name of an effect preset based on its index within an effect presets file.

For further details on these functions, see the *Scream Library Reference* documents.

Working with INI-Format Effect Preset Files

Effect preset files in the `INI` format can contain preset definitions for buss effects on the NGS or NGS2 synthesizers. However, Scream Tool supports exporting of `INI`-format presets files for I3DL2 reverb effects only. `INI`-format preset files for the following effect types must be created manually:

- NGS2 delay and chorus auxiliary buss effects
- pre-master submix buss effects
- master buss effects

For further details, see Creating INI-Format Presets Files below.

Loading an Effect Preset File

You can load INI-format presets files into Scream from disk or from memory using the following functions:

- sceScreamPresetFileLoad()
  Loads an INI-format presets file from disk
- sceScreamPresetFileLoadFromMem()
  Loads an INI-format presets file from memory

Both these functions return a preset file handle (SceScreamIniHandle), which you reference when applying a preset from an INI file.

To unload an INI-format presets file and release its memory, use the sceScreamPresetFileUnload() function.

Applying an Effect Preset

Having loaded an INI-format effect preset file, you can apply presets it contains for auxiliary buss or pre-master submix buss effects using the following functions:

- NGS and NGS2 legacy effects behavior: sceScreamReverbSetCustomPreset()
  Applies a reverb preset, referenced by index.
- NGS and NGS2 legacy effects behavior: sceScreamReverbSetCustomPresetByName()
  Same as above, except referencing a preset by name.
- NGS2 effect APIs enabled: sceScreamEffectSetCustomPreset()
  Applies an auxiliary buss preset, referenced by index.
- NGS2 effect APIs enabled: sceScreamEffectSetCustomPresetByName()
  Same as above, except referencing a preset by name.
- sceScreamPremasterSubmixSetCustomPreset()
  Applies a pre-master submix buss effect preset, referenced by index, on the NGS2 or NGS synths.
- sceScreamPremasterSubmixSetCustomPresetByName()
  Same as above, except referencing a preset by name.

In each case, you identify the target auxiliary effect instance or pre-master submix index in the functions' first argument. In the second argument you specify the SceScreamIniHandle of an INI file in which the desired preset is stored. And in the functions' third argument you identify a particular preset to apply, either by index or name reference.

For further details on applying presets from INI files, see:

- Working with Custom Effect Presets in the "Working with Auxiliary Buss Effects" chapter
- Working with Custom Presets in the "Working with Pre-Master and Master Signal Processors" chapter

Querying Effect Presets

You can obtain a count of the number of preset definitions contained within a presets file using the sceScreamPresetFileGetPresetCount() function. You pass in the SceScreamIniHandle of a presets file to query, and the function returns a count of the number of presets it contains. When setting a custom preset by index, using the sceScreamReverbSetCustomPreset() or sceScreamPremasterSubmixSetCustomPreset() functions, this count (minus 1) serves as the maximum value to use for the *presetIndex* parameter. Note that because the *presetIndex* parameter to these functions expects a zero-based index of presets, you must subtract one (1) from the value returned by the sceScreamPresetFileGetPresetCount() function to determine the maximum index value.

You can obtain the names of presets contained in a presets file using the sceScreamPresetFileGetPresetName() function. You pass in the SceScreamIniHandle of a presets file to query, and the index of the preset for which to retrieve a name. You must also specify a character buffer in which to store the preset name and its length. When setting a custom preset by name, using the `sceScreamReverbSetCustomPresetByName()` or `sceScreamPremasterSubmixSetCustomPresetByName()` functions, you can use the output of the `sceScreamPresetFileGetPresetName()` function as a value for the *`presetName`* parameter. For further details, see the *Scream Library Reference* documents.

Creating INI-Format Presets Files

`INI` is a simple text-based file format used to define property values. The structure of an `INI` file is comprised of preset names (in square brackets) and lists of property values. You can create presets `INI` files for the NGS and NGS2 synthesizers' I3DL2 reverb and premaster submix buss effects using a text editor. The syntax is very simple. However, it is important that property names are referred to correctly. The following is an example `INI` file that contains a single preset for the premaster submix buss compressor effect.

```
[compressor_preset_1]
compEffectOn = 0
compLinkedChannels = 1
compPeakMode = 0
compThresholdDB = -6.0
compRatio = 2.0
compAttackTimeMS = 10
compReleaseTimeMS = 60
compMakeupGainDB = 3.0
compSoftKneeDB = 0.0
```

The first line defines the name of the preset ("`compressor_preset_1`"). Following that is a list of compressor properties and values. Presets must have unique names, and be separated by a blank line. The key names are the names of the members of the `SceScreamSndPremasterSubmixProps` structure.

By way of a further example, the following code shows an `INI` file representation of an I3DL2 reverb preset.

```
[rev-generic]
Room = 0
Room_HF = -100
Decay_time = 1.49
Decay_HF_ratio = 0.83
Reflections = -2602
Reflections_delay = 0.007
Reverb = 200
Reverb_delay = 0.011
Diffusion = 100
Density = 100
HF_reference = 5000
EarlyReflectionPatternRoomLeft = 0
EarlyReflectionPatternRoomRight = 1
EarlyReflectionScalar = 100
LF_reference = 80
Room_LF = 0
DryMB = -10000
```

The key names are the names of the members of the `SceScreamSndReverbProps` structure, except for the *`EarlyReflectionPattern`* member, which has two elements. These two elements are specified by the names *`EarlyReflectionPatternRoomLeft`* and *`EarlyReflectionPatternRoomRight`* and their values are those of the early reflections pattern constants, which are described in "I3DL2 Reverb Early Reflections Patterns" in the *Scream Library Reference* documents.

12 Working with Mix Snapshots

Mix snapshots capture Group volume and output destination settings that can be recalled at run time, enabling advanced control over mixing. Designers can create group mixer files in Scream Tool that contain settings for an initial foundation mix, the mixer base level (headroom control), plus a number of mix snapshots. You can load a group mixer file at initialization time. Thereafter, you can activate, deactivate, and query mix snapshots at run time using various function calls.

Understanding Mix Snapshots

To make effective use of mix snapshots it is important to understand operational details of the snapshots mechanism, and how it interacts with the Group volumes that can be set with the `sceScreamSetMasterVolume()` function; see Setting and Retrieving Group and Master Volumes in the "Working with Groups" chapter.

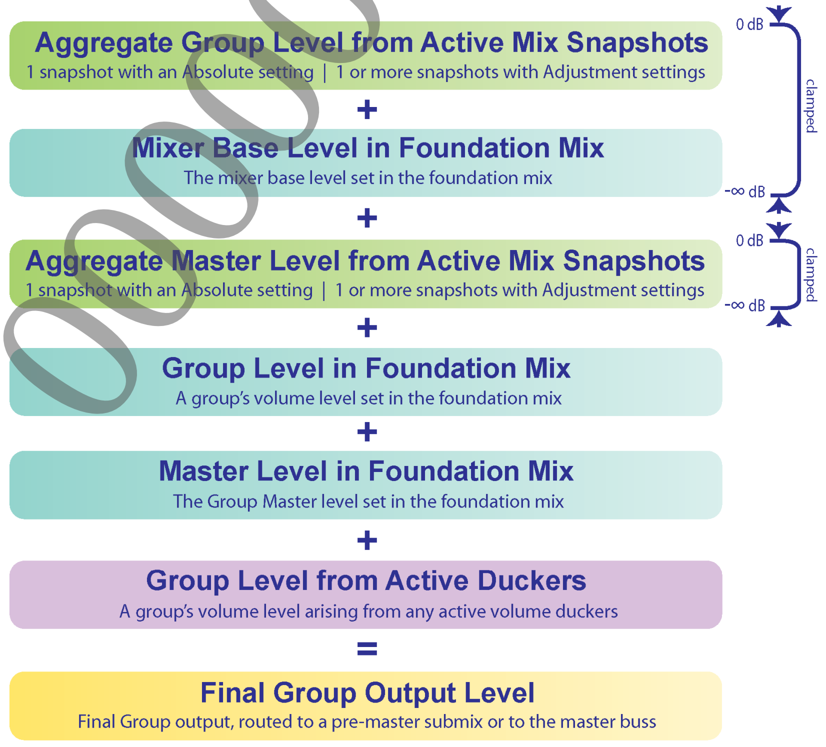
Group Volume Level Components

When working with mix snapshots, it is important to understand the Group volume components that comprise a Group's final output level and how they combine. These volume components are as follows:

- The aggregate level for a Group set by any active mix snapshots
- The level set for a Group in the foundation mix
- The aggregate Group Master level set by any active mix snapshots
- The foundation mix base level
- The foundation mix Group Master level
- The level set for a Group by any active volume duckers

Working in units of decibels (dB), Group volume components add together to produce a final output level for each Group as shown in Figure 16.

Figure 16 Group Volume Components


Aggregate Group Level from Active Mix Snapshots
1 snapshot with an Absolute setting | 1 or more snapshots with Adjustment settings
+
Mixer Base Level in Foundation Mix
The mixer base level set in the foundation mix
+
Aggregate Master Level from Active Mix Snapshots
1 snapshot with an Absolute setting | 1 or more snapshots with Adjustment settings
+
Group Level in Foundation Mix
A group's volume level set in the foundation mix
+
Master Level in Foundation Mix
The Group Master level set in the foundation mix
+
Group Level from Active Duckers
A group's volume level arising from any active volume duckers
=
Final Group Output Level
Final Group output, routed to a pre-master submix or to the master buss
0 dB
clamped
-∞ dB
0 dB
clamped
-∞ dB

Some examples of how Group volume levels aggregate in different scenarios are detailed in the following table (**Note**: read down the table columns, not across the rows).

Table 31 Group Level Example

Volume Component	Scenario 1: Group Level	Scenario 2: Group Level	Scenario 3: Group Level	Scenario 4: Group Level
Active Mix Snapshot(s): Group Level	Inactive	Inactive	Absolute: +6 dB	Adjustment: +4 dB Adjustment: -9 dB
Foundation Mix: Base Level	-3 dB	-3 dB	-3 dB	-3 dB
Active Mix Snapshot(s): Master Level	Inactive	Inactive	Inactive	Inactive
Foundation Mix: Group Level	0 dB	-6 dB	-6 dB	-6 dB
Foundation Mix: Master Level	0 dB	-3 dB	-3 dB	-3 dB
Active Ducker(s): Group Level	Inactive	Inactive	Inactive	Inactive
Final Group Output	**-3 dB**	**-12 dB**	**-6 dB**	**-17 dB**

The Foundation Mix

The foundation mix is the set of Group volumes that can be set, individually, using the `sceScreamSetMasterVolume()` function. A group mixer file includes foundation mix settings, enabling audio designers to define initial settings for every Group (volume levels and output destinations) and for the Group Master. The foundation mix is the state of the mixer when no mix snapshots are active, and includes the following components:

- The mixer base level setting (used for headroom control)
- Volume levels for all Groups and the Group Master
- Output destinations for all Groups

Consultation Point: If your audio designers are including group output destination settings as part of a foundation mix, make sure that Sounds inherit output destinations from the group to which they are assigned. This requires setting the *`outputDest`* parameter to `SCE_SCREAM_SND_OUTPUT_DEST_BY_GROUP` when starting Sounds using the `sceScreamPlaySoundByIndexEx()` or `sceScreamPlaySoundByNameEx()` functions. Otherwise output destination settings included in a group mixer file are not honored at run time.

Note: Changing Group volume settings using the `sceScreamSetMasterVolume()` function effectively alters the foundation mix from the state that was established when initializing Scream with a group mixer file. Consequently, it may be prudent to store the initial Group volume settings (obtained from calling `sceScreamGetMasterVolume()`) that comprise the foundation mix, so that you can reset Groups to their initial volumes. Otherwise, the foundation mix, as conceived by your audio designers, may diverge from the realities at run time. Given the interaction between foundation mix and snapshot settings, changing the foundation mix could offset the effects of activating mix snapshots in unpredictable and perhaps undesirable ways.

Snapshot Setting Modes

Mix snapshots effect changes to Group volumes using a choice of modes, set at design time in Scream Tool. The snapshot setting modes are:

- **Inactive** (default) – Group volume level is not included in the snapshot. When a snapshot is activated in which a Group is set to Inactive, the Group's volume is unaffected, remaining at its existing setting.

- **Absolute** – Group volume level is set by a single snapshot. If multiple snapshots contain Absolute settings for a Group, the snapshot with the highest priority prevails. When a snapshot is activated in which a Group has an Absolute setting, the Group's volume transitions to its new level over the snapshot's transition-in time. For example, suppose the Music Group level is at -12 dB in the foundation mix. A snapshot is activated that sets the Music Group to Absolute -18 dB. The Music Group transitions from -12 dB to -30 dB over the snapshot's transition-in time. Note that additional Group volume components, such as the foundation mix Master level and the mixer base level, also apply.
- **Adjustment** – Group volume level may be set by a combination of multiple snapshots. If multiple snapshots contain Adjustment settings for a Group, their settings add together to produce a combined snapshot level for the Group. When a snapshot is activated in which a Group has an Adjustment setting, the Group's volume transitions to its new level over the snapshot's transition-in time. If multiple snapshots are active, Group volume levels aggregate Adjustment settings applied to each Group from the respective snapshots. For example, the Music Group level is at -12 dB in the foundation mix, and a snapshot is activated that sets it to Adjustment -18 dB. The Music Groups transitions from -12 dB to -30 dB over the snapshot's transition-in time. Then, another snapshot is activated that sets the Music Group to Adjustment +10 dB. The Music Groups transitions from -30 dB to -20 dB over the second snapshot's transition-in time. Note that additional Group volume components may also apply.

See Combining Mix Snapshots and Group Volume Level Components for further details.

Combining Mix Snapshots

Multiple mix snapshots can be active at the same time. When multiple snapshots are active, Group volume settings from different snapshots combine according to some basic rules. Layering snapshots allows you to make temporary adjustments to a mix as demanded by shifting game contexts. For example, you might need to temporarily reduce SFX levels to allow critical dialog lines to be heard. When combining multiple snapshots, the rules for determining a Group's volume level are as follows:

- **Absolute settings preempt Adjustment settings.** If a Group has an Absolute setting from one snapshot and Adjustment settings in other snapshots, the value from the one Absolute setting is used, and the Adjustment settings are ignored.
- **The Absolute setting with the highest priority prevails.** If a Group has Absolute mode settings in multiple snapshots, the Absolute setting that belongs to the snapshot with the highest priority is the setting that is used.
- **Multiple Adjustment settings are aggregated.** If a Group has Adjustment settings in multiple snapshots, these settings are aggregated to produce a final Adjustment value, which is further offset by the Group's foundation mix level and by the mixer base level.

Transitioning Mix Snapshots

Snapshot transition-in and transition-out times allow smooth volume level transitions when activating and deactivating mix snapshots. On activation of a snapshot, calculations are made to determine new volume levels for each Group. See Group Volume Level Components above for further details. Group volumes then transition from current levels to target levels over the time set as the activated snapshot's transition-in time.

When a snapshot is de-activated, similar calculations are made to determine target volume levels for each Group. Each Group then transitions to its target level over the time set as the de-activated snapshot's transition-out time.

Output Destinations and Mix Snapshot Transitions

Designers can include group output destination settings in mix snapshots. Note, however, that Sound output destinations default to the master buss, regardless of the output destination of the group to which a Sound is assigned. To override this default, you can set Sound output destinations to inherit those of the group to which Sounds are assigned. This setting is also necessary for snapshot-specified group output destinations to be honored at run time.

Consultation Point: If designers are including group output destination settings in mix snapshots, make sure that Sounds inherit output destinations from the group to which they are assigned. This requires setting the *`outputDest`* parameter to `SCE_SCREAM_SND_OUTPUT_DEST_BY_GROUP` when starting Sounds using the `sceScreamPlaySoundByIndexEx()` or `sceScreamPlaySoundByNameEx()` functions.

The Scream Runtime configures Sound output destinations at the time a Sound is started (that is, on calling the `sceScreamPlaySoundByIndexEx()` or `sceScreamPlaySoundByNameEx()` functions). A consequence of this mechanism, in relation to snapshot-specified group output destinations, is that a Sound continues to output to the destination to which it was initially routed by the function – even if the output destination, inherited from a group to which the Sound is assigned, changes while the Sound is playing by activating a snapshot. However, a Sound that is started after the output destination (for the group to which the Sound is assigned) has changed is routed to the group's new output destination. This mechanism is beneficial, as it prevents audio discontinuities that might otherwise result from changing output destinations while Sounds are playing.

Snapshot Priorities

In a case where multiple snapshots are activated that contain Absolute settings for a given Group, the snapshot with the highest priority prevails. You can retrieve snapshot priorities using the `sceScreamGetMixSnapshotPriority()` function.

Setting the Group Mixer Update Rate

The group mixer update rate (or perhaps more precisely, update *interval*) defines the group mixer processing granularity. You set the group mixer update rate when initializing Scream, using the `SceScreamPlatformInitEx2` structure's *`groupMixerUpdateRate`* member. Values are expressed in fractions of a second, within the range 0.0 to 1.0. If you are not using mix snapshots in your game, you can set the *`groupMixerUpdateRate`* member to 0.0, which effectively disables the mix snapshots mechanism. Left unset, the *`groupMixerUpdateRate`* member defaults to 0.0167 seconds (equivalent to 4 Scream ticks). Note that custom *`groupMixerUpdateRate`* values are rounded to the nearest Scream tick (1/240th of a second). If this rounded number is *n*, Group mixing is updated every *n*th Scream tick.

Loading a Group Mixer File

To use mix snapshots, you must obtain a group mixer file from your audio designer. Designers can create mix snapshots in Scream Tool, and export platform-specific binary group mixer files (with a GMX extension) containing snapshot definitions, as well as foundation mix settings.

You pre-load a group mixer file into memory, and then provide the corresponding memory pointer as a value for the `SceScreamPlatformInitEx2` structure's *`pGroupMixerFile`* member when initializing Scream. This initializes all Group levels, the mixer base level, and the Group Master level per the foundation mix settings in the group mixer file.

Constraining the Number of Active Snapshots

The mix snapshots mechanism supports multiple snapshots being active at the same time. You can set an upper limit for the number of simultaneously active snapshots at initialization time using the `SceScreamPlatformInitEx2` structure's *`maxActiveSnapshots`* member, which defaults to 32. Note that setting this member to zero effectively disables the group mixer system.

Activating Mix Snapshots

You activate a mix snapshot by calling the `sceScreamActivateMixSnapshot()` function. In addition to activating a snapshot, the function allows you to scale the amount of Group level setting defined by the snapshot, and to override the snapshot's transition-in time.

In the function's *name* parameter, you reference the name of a snapshot to activate, as defined in a group mixer file loaded at initialization time.

The *mixScalar* parameter defaults to 1.0, allowing the snapshot's level-setting impact to be as defined by the designer. Setting *amount* lower than 1.0 reduces the level-setting impact of a snapshot. For example, if the level set in a snapshot for the SFX Group is 6.0 dB, and you set *amount* to 0.5, the level-setting impact of the snapshot for the SFX Group is reduced to 3.0 dB.

Similarly, the function's *fadeTimeOverride* parameter defaults to -1.0, allowing the snapshot's transition-in time to be as defined by the designer. Setting *fadeTimeOverride* to greater than zero overrides the original transition-in time. For example, if the transition-in time set for a snapshot is 1.5 seconds, and you set *fadeTimeOverride* to 0.5, the transition-in time for the snapshot changes to 0.5 seconds.

Deactivating Mix Snapshots

You can deactivate a single snapshot or deactivate all active snapshots using the `sceScreamDeactivateMixSnapshot()` and `sceScreamDeactivateAllMixSnapshots()` functions. If deactivating all snapshots, you must specify a transition-out time to apply to all deactivating snapshots in the `sceScreamDeactivateAllMixSnapshots()` function's *fadeTimeSeconds* parameter.

In the `sceScreamDeactivateMixSnapshot()` function's *name* parameter, you reference the name of a snapshot to deactivate. You can also optionally override the snapshot's transition-out time by setting a positive value, expressed in seconds, for the function's *fadeTimeOverride* parameter.

Querying Mix Snapshots

You can verify whether a snapshot is active, retrieve a count of the number of active snapshots, and retrieve the priority set on a snapshot.

Verifying Whether a Snapshot is Active

You can verify whether a particular mix snapshot is active using the `sceScreamIsMixSnapshotActive()` function. You pass in the name of a snapshot to query, and the function returns a Boolean value. `TRUE` indicates that the specified snapshot is active; `FALSE` indicates that the snapshot is inactive.

Retrieving a Count of Active Snapshots

You can retrieve a count of the current number of active snapshots using the `sceScreamGetActiveMixSnapshotCount()` function. The function takes no arguments and simply returns a count of the number of active mix snapshots, in the range: 0 to (the value set at initialization time for) `SceScreamPlatformInitEx2.maxActiveSnapshots`.

Retrieving a Snapshot's Priority

You can retrieve the priority value set on a snapshot using the `sceScreamGetMixSnapshotPriority()` function. You pass in the name of a snapshot to query, and specify a `uint32_t` variable in which to receive the snapshot's priority. The function returns a Boolean value indicating the success of the operation.

Setting the Mixer Base Level

The group mixer base level provides an adjustment control on all Group volume levels, before further scaling by the Group Master level. It is conceived as an additional headroom control, and is part of the foundation mix settings included in a group mixer file. See Figure 16, Group Volume Components above.

You can also set the base level at run time using the `sceScreamSetGroupMixerBaseLevel()` function. This provides a way to effect volume level changes relatively, simultaneously, and (usually) temporarily, across all Groups.

Note: Like setting Group volumes with the `sceScreamSetMasterVolume()` function, setting the base level alters the foundation mix, and thus may offset the effects of activating mix snapshots in unpredictable ways. When setting the mixer base level, it is prudent to store its initial value, so that you can reset to this value later, allowing your audio designer's foundation mix settings to remain intact.

The `sceScreamSetGroupMixerBaseLevel()` function takes a target base level setting, expressed in dB, and a fade time that specifies, in seconds, the amount of time for the base level to reach its target.

13 Working with CCSounds

A CCSound (short for Continuous Controller Sound) is a special type of Scream Sound that enables continuous, dynamic control over the properties of scripted Sounds based on one or more input parameters, which may be local or global in scope. Designers can include CCSounds in Bank contents. When loaded into the Scream Runtime, local and global variables (not to be confused with local and global registers) corresponding with input parameters associated with CCSounds are automatically created.

CCSounds map one or more input parameters to Sound properties through curves. By manipulating the values of these input parameters (local and/or global variables), a CCSound can control base properties of scripted Sounds, plus associated synthesizer voice properties, such as pre-send filter, auxiliary send, and distortion settings. A CCSound can also start, stop, and crossfade between scripted Sounds based on values of its input parameter(s).

Scream provides APIs by which you can initialize Scream for CCSound usage, and manipulate and query local and global variables.

For more information on CCSounds, see “CCSound Editor” in the *Scream Tool Help*.

Initializing Scream for CCSound Usage

You can set the CCSound processing granularity, constrain the number of simultaneously active CCSounds, and constrain the number of global variables.

Setting the CCSound Update Rate

The CCSound update rate (or perhaps more precisely, update *interval*) defines CCSound processing granularity. You set the CCSound update rate when initializing Scream using the `SceScreamPlatformInitEx2` structure's *`ccSoundUpdateRate`* member. Values are expressed in fractions of a second, within the range 0.0 to 1.0. If you are not using CCSounds in your game, you can set the *`ccSoundUpdateRate`* member to 0.0, which effectively disables the CCSound mechanism. Left unset, the *`ccSoundUpdateRate`* member defaults to 0.0084 seconds (equivalent to 2 Scream ticks). Note that custom *`ccSoundUpdateRate`* values are rounded to the nearest Scream tick (1/240th of a second). If this rounded number is *n*, CCSounds are updated every *n*th Scream tick.

Constraining the Number of CCSounds

The CCSound mechanism supports multiple CCSounds being active at the same time. You can set an upper limit for the number of simultaneously active CCSounds at initialization time using the `SceScreamPlatformInitEx2` structure's *`maxCCSounds`* member, which defaults to 32. Note that setting this member to zero effectively disables the CCSound mechanism. Note that the memory cost of each simultaneously active CCSound is 2 kB. The overall CCSound memory cost with the default value for *`maxCCSounds`* is therefore 64 kB.

Constraining the Number of Global Variables

You can constrain the number of simultaneously existing global variables. You do this at initialization time using the `SceScreamPlatformInitEx2` structure's *`maxGlobalVariables`* member, which defaults to 64.

Note that the maximum number of local variables per CCSound is constrained by the Scream system constant `SCE_SCREAM_SND_MAX_LOCAL_VARIABLES`, the value of which is 16.

Understanding a CCSound's Associated Variables

CCSounds may contain complex and intricate mappings between their input parameters and scripted Sounds they reference, including:

- key-on/key-off settings
- crossfade settings between scripted Sounds
- mappings to base properties of scripted Sounds (volume, pitch, pan azimuth, pan focus, and so on)
- synthesizer parameter override settings
- mappings to synthesizer parameter adjustments
- complex multi-segment, multi-shape curves

While the input parameters associated with CCSounds are saved with Bank contents, and corresponding local and global variables are automatically created on loading such Banks into the Scream Runtime, a programmer's understanding of the usage of CCSound input parameters is entirely dependent upon communication with the audio designer responsible for their creation. For successful use of CCSounds, audio designers must supply the following information to audio programmers:

- The names of input parameters associated with each CCSound, and any expected initial values.
- The purpose of each input parameter, and the corresponding game state with which it should be driven.
- Which input parameters to a CCSound are local and which are global.

Local and Global Variables

The variables associated with a CCSound (based on its design-time input parameters) may be local or global in scope. The values of local variables are unique to each CCSound. For example, a local variable named *speed* can have different values with respect to each CCSound in which it is used. The values of global variables are shared between CCSounds. For example, a global variable named *weather* has the same value for each CCSound that references it.

CCSound local and global variables (and their corresponding design-time input parameters) are normalized to the range 0.0 to 1.0. Programmers must ensure that game states are appropriately mapped to this normalized range. For example, suppose your audio designer creates a CCSound to produce vehicle sounds, which uses an input parameter named *speed* to represent the current speed of a vehicle. The game must continuously update the corresponding *speed* variable on game play, interpolating the actual game speed of a vehicle to fall within the normalized range of the *speed* variable, using a formula such as the following:

$$\text{Normalized speed} = \frac{\textit{current game speed of vehicle}}{\textit{maximum game speed of vehicle}}$$

Playing CCSounds

You play CCSounds in exactly the same way as scripted Sounds. That is, using the `sceScreamPlaySoundByIndexEx()` and `sceScreamPlaySoundByNameEx()` functions. If specified by your audio designer, before playing a CCSound, you can prime the values of its associated local and global variables, which otherwise default to 0.0.

The Scream CCSound sample program illustrates playing a CCSound and setting its input variables.

Working with CCSound Local Variables

You can set local variables associated with a CCSound individually or collectively, and you can retrieve their values individually. You reference local variables by a hash, based on a variable's name. You can derive name hashes using the `sceScreamGetHashFromName()` function. You can then use the returned hash values as input to the `sceScreamSetLocalVariableByHash()` and `sceScreamGetLocalVariableByHash()` functions' *`namehash`* parameter (when setting or retrieving an individual local variable), or the `SceScreamSndLocalVarData` structure's *`namehash`* member (when setting local variables collectively).

Setting Local Variables Individually

To set an individual local variable, you use the `sceScreamSetLocalVariableByHash()` function. In the function's *`handle`* parameter, you specify the handle of a CCSound upon which to set a local variable. For the function's *`namehash`* parameter, you identify the local variable to set using a 32-bit hash based on the local variable's name, derived from the `sceScreamGetHashFromName()` function. In the function's *`val`* parameter you specify a floating-point value to set the local variable to.

Setting Local Variables Collectively

To set multiple CCSound local variables you use the `SceScreamSndLocalVarData` structure. This allows you to collectively set initial values before playing a CCSound or while a CCSound is playing. The `SceScreamSndLocalVarData` structure allows you to set a variable number of local variables, according to how many are associated with a particular CCSound, and up to a maximum of `SCE_SCREAM_SND_MAX_LOCAL_VARIABLES` (16) per CCSound. The `SceScreamSndLocalVarData` structure is embedded as the *`localVariableData`* member of the `SceScreamSoundParams` structure. And, in turn, the `SceScreamSoundParams` structure is used both when:

- playing a CCSound – as the expected data type for the `sceScreamPlaySoundByIndexEx()` and `sceScreamPlaySoundByNameEx()` functions' *`params`* parameter
- updating CCSound parameters – as the expected data type for the `sceScreamSetSoundParamsEx()` function's *`params`* parameter

The `SceScreamSndLocalVarData` structure includes *`namehash`* and *`val`* members, which are used in exactly the same manner as the corresponding parameters of the `sceScreamSetLocalVariableByHash()` function. For further details, see the `SceScreamSndLocalVarData` structure in the *Scream Library Reference* documents.

Setting Local Variables on Child Sounds

Designers can create scripted Sounds that start CCSounds as child Sounds, using the *Start Child Sound* Grain. Setting a local variable on the parent Sound handle sets any similarly named local variables on associated child Sounds. This mechanism enables an application to control a child CCSound even without having a handle to the child CCSound itself.

Retrieving the Value of Local Variables

To retrieve the value of a local variable, you use the `sceScreamGetLocalVariableByHash()` function. In the function's *`handle`* parameter you specify the handle of a CCSound from which to retrieve a local variable value. For the function's *`namehash`* parameter, you identify the local variable to retrieve using a 32-bit hash based on the local variable's name, derived from the `sceScreamGetHashFromName()` function. In the function's *`val`* parameter, you point to a floating-point variable in which to receive the value of the specified local variable.

Note: Using CCSound Editor in Scream Tool, designers can set slewing properties that constrain the rate of change of input parameters. However, local variable values retrieved using the `sceScreamGetLocalVariableByHash()` function represent the last value set, and not necessarily the current slewed value.

Working with Global Variables

Scream uses a reference counting algorithm to manage global variables, which takes account of global variable references both from loading and unloading of Banks and from API calls. You can retrieve the current number of global variables, add and delete global variables, retrieve global variable indices by name and by hash, and set and retrieve global variable values.

Retrieving the Number of Global Variables

Addition and deletion of global variables is dynamic, and can be initiated both by API calls and by loading and unloading of Banks. You can retrieve the number of global variables that currently exist in the system using the `sceScreamGetNumGlobalVariables()` function, which simply returns the number of global variables. The maximum number of global variables is constrained at initialization time by the `SceScreamPlatformInitEx2` structure's *`maxGlobalVariables`* member, and defaults to `SCE_SCREAM_SND_DEFAULT_MAX_GLOBAL_VARIABLES` (64).

Adding and Deleting Global Variables

Global variables are added and deleted automatically upon loading and unloading Banks that contain global variables. You can also add and delete global variables explicitly using the `sceScreamAddGlobalVariable()` and `sceScreamDeleteGlobalVariable()` functions. Adding global variables explicitly can be expeditious as it allows you to set their values prior to loading corresponding Banks containing CCSounds that reference them.

When adding a global variable using the `sceScreamAddGlobalVariable()`function, you can optionally specify a pointer in which to store an index associated with the variable. The index is valid whether or not the global variable existed prior to calling the `sceScreamAddGlobalVariable()` function. In addition to the optional pointer, you must also specify a name to be assigned to the added global variable.

To explicitly delete a global variable you use the `sceScreamDeleteGlobalVariable()` function. You specify the name of a global variable to delete, and the function returns `TRUE` or `FALSE` indicating success of the delete operation.

You can add a global variable and set its initial value using the `sceScreamAddSetGlobalVariable()` function. The function is similar to the `sceScreamAddGlobalVariable()` function, but includes an additional parameter, *`val`*, in which you specify the value to set on the global variable being added.

Obtaining Global Variable Indices

You reference global variables, when retrieving or setting their values, using a global variable index. You obtain global variable indices upon adding a global variable to the system with the `sceScreamAddGlobalVariable()` and `sceScreamAddSetGlobalVariable()` functions. Both functions provide an *`outVarIndex`* parameter, allowing you to specify an optional pointer in which to store the index associated with the specified global variable.

Setting and Retrieving Global Variable Values

You can set and retrieve global variable values using the `sceScreamSetGlobalVariableByIndex()` and `sceScreamGetGlobalVariableByIndex()` functions. Both functions require a global variable index, by which you reference a specific global variable. See Obtaining Global Variable Indices above for further details.

To set the value of a global variable, you use the `sceScreamSetGlobalVariableByIndex()` function, specifying the global variable's index and a floating-point value to set. To retrieve a global variable value, you use the `sceScreamGetGlobalVariableByIndex()`function, specifying the global variable's index, and a pointer to a floating-point variable in which to store the retrieved value.

You can also set a global variable's initial value while adding it to the system using the `sceScreamAddSetGlobalVariable()` function.

CCSounds and Looping Designation

Similarly to scripted Sounds, designers can set the Looping property for CCSounds to a Boolean value in Bank contents. With scripted Sounds, this Looping designation merely indicates to the programmer the likely behavior of a Sound. Sounds with Looping set to `TRUE` (whether due to the presence of looping assets or a looping script) are assumed to continue until explicitly stopped, whereas Sounds with Looping set to `FALSE` are assumed to be one-shots that will stop themselves.

With CCSounds, the Looping designation actually effects run time playback behavior. CCSounds that control scripted Sounds with looping assets – such as typical vehicle engine assets, for example – are essentially looping CCSounds. They continue playing as long as one or more input parameter values are within the keying range of one or more constituent scripted Sounds, and must be stopped explicitly. Designers can also set a CCSound's Looping property to `FALSE`, making it a one-shot CCSound. One-shot CCSounds stop themselves as soon as there are no active scripted Sounds. That is, no input parameter value(s) are within the keying range of any constituent scripted Sounds, and any previously triggered Sounds have naturally concluded.

Note: One-shot CCSounds stop themselves immediately after starting if initial input parameter values fall outside the keying range of constituent scripted Sounds, and if the keying options on such scripted Sounds do not include *Key On With CCSound*.

14 Simulating Complex Environmental Audio

This chapter discusses some advanced techniques for simulating complex environmental audio.

Obstruction and Occlusion

Obstruction and occlusion describe the rendering of various types of acoustic barriers between the listener and sound source. Sound travels through and around barriers in a variety of scenarios, including those where the listener and sound source are in the same and in different acoustic spaces. For example, sound travels:

- around objects and corners; listener and sound source in same space
- through open and closed doors and windows; listener and sound source in different spaces
- through walls from one room to another; listener and sound source in different spaces

Barrier-transmitted sounds are different from their unimpeded counterparts. The ability to simulate these effects can significantly enhance the realism of a virtual environment in which the listener is placed.

Auditory Effects of Acoustic Barrier Transmission

Sound transmitted through acoustic barriers undergoes both frequency-dependent and frequency-independent attenuation. The frequency-dependent attenuation has a low-pass character. The amount of attenuation varies with the material, thickness, and construction of the barrier. Sound traveling around corners or through open doors and windows also undergoes low-pass attenuation due to the refraction of component sound waves. With refraction, the dependency is the relationship between the wavelengths of frequency components and the size of the opening (or angle of the corner). Lower frequency components have longer wavelengths and are less attenuated by refraction. Higher frequency components have shorter wavelengths and are more attenuated by refraction.

A critical distinction regarding barrier-transmitted or refracted sounds is whether the listener is in the same acoustic space as the sound or in a different space. If the listener is in the same space, attenuation applies only to the direct sound. That is, reverberation within the same space is unaffected by any barriers between listener and sound source. The simulation of barrier transmission where listener and sound source are in the same space is called obstruction.

If the listener is in a different space from the sound source, attenuation applies to both the direct sound and its reverberation, regardless of whether the barrier is solid (such as a wall or closed door) or is a constrained opening (such as an open window or door). The simulation of barrier transmission where listener and sound source are in different spaces is called occlusion.

Simulating Obstruction

To simulate the audio effects of obstruction, you can apply the post-send filter on the obstructed Sounds. Because obstruction does not change the reverberated component of the Sound, you should not use the pre-send filters when simulating obstruction, as they modify the Sound before its signal is sent to a reverb instance. You set parameter values for the post-send filter using the `SceScreamSndIIRFilterParams` structure. Select a filter type (a low-pass may be the most appropriate), and set suitable cut-off frequency and resonance values that approximate the acoustic properties of a barrier in your game. For further details on configuring and applying post-send filter settings on a Sound, see the "Working with Pre- and Post-Send Filters and Effects" chapter.

15 Working with Distance Models, Doppler, and Spatialization

This chapter explains how to work with distance models, Doppler pitch shifting, and spatial positioning of Sounds. These features enhance Sound playback with listener perspective. They require programmers to set positional coordinates for a Sound and for a referenced listener. From these respective coordinates, and time deltas between successive updates, Scream calculates the relative distance, speed, and angle of a Sound from the perspective of a listener. Scream supplies the distance value to a Sound's assigned distance model; the speed calculation to the internal Doppler effect; and programmers can apply the calculated angle to a Sound's azimuth setting.

Distance Models

Distance models simulate the auditory effects of distance between a sound source and listener. A distance model defines a set of attenuation curves assigned to five parameters, significant in the simulation of distance:

- Dry level
- Wet level
- LFE level
- Air Absorption; targets a pre- or post-send filter
- Spread; targets focus

When designers assign a distance model to a Sound, by default its settings are applied to every asset Grain contained within the Sound's script. Asset Grains (*Waveform* and *Stream* Grains) can also override their containing Sound's distance model settings.

Like buss presets and mix snapshots, distance models are not included in Bank contents. Instead, designers export distance models to a runtime binary format for deployment in a game.

The information in this section explains how to work with distance models from an audio programmer perspective. For further information about distance models from an audio designer perspective, see the "Distance Model Editor" chapter in the *Scream Tool Help* document.

See the Scream Distance Model example program for an illustration of distance models in use.

Loading a Distance Model File

To use distance models in Scream you must pre-load a distance model file (having a `DML` extension) into memory, and then provide the corresponding memory pointer as a value for the `SceScreamPlatformInitEx2` structure's *`pDistanceModelFile`* member when initializing Scream.

Supplying Distance Input

Sounds with an assigned distance model require dynamic distance input to operate. However, there is no *distance* parameter. Rather, you specify distance in terms of a Sound's 3D positional coordinates with respect to those of a referenced listener. Based on these coordinates for Sound and listener position, Scream calculates distance and applies it as input to a Sound's assigned distance model.

You set 3D coordinates on a Sound using the `SceScreamSoundParams` structure's *`position`* member. And you reference a listener using the `SceScreamSoundParams` *`listenerHandle`* member. A listener, in this context, is a proxy for a listening point in game world space. For example, this might be a game character or the location of a camera.

You create listeners using the `sceScreamCreateListener()` function. Scream supports up to `SCE_SCREAM_SND_MAXLISTENERS` (16) simultaneous listeners. Having created a listener you must set and dynamically update its position using the `sceScreamSetListener()` function. You specify the handle of a listener (returned by the `sceScreamCreateListener()` function) upon which to set coordinates in the `sceScreamSetListener()` function's *`handle`* parameter. And you specify the listener's position in game-world space in the *`location`* parameter, using a `SceScreamSnd3DVector` structure.

Note: For distance model processing, only the location of a listener is required, not the listener's orientation. Consequently, the `sceScreamSetListener()` function's *`front`* and *`top`* parameters are ignored.

Setting the Target Air Absorption Filter

The distance model Air Absorption element can target a choice of a Sound's filters or can be disabled. By default, the Air Absorption element targets a Sound's post-send filter. The post-send filter is intended primarily for use as a rendering filter (see Simulating Obstruction), and cannot be set by designers in Bank contents. If you are not using Sound post-send filters for rendering, this could be a good choice as it is guaranteed not to conflict with filter settings in Bank contents. However, if you *are* using post-send filters for rendering, you have two further options:

- Apply Air Absorption filtering to the 4th pre-send filter (that is, `SceScreamSynthParams.`*`preSendFilter3`*)
- Disable Air Absorption filtering

You set these options as Sound flags, applying one of the following to the `SceScreamSoundParams` structure's *`flags`* member:

- `SCE_SCREAM_SND_FLAG_DIST_MODEL_PRESEND_FILTER_3`
- `SCE_SCREAM_SND_FLAG_DIST_MODEL_NO_FILTER`

Consultation Point: If using a Sound's *pre-send filter 3* for distance model air absorption, check with your audio designer to ensure there are no conflicts with Bank contents. Note that in Scream Tool, the pre-send filters are numbered from 1, so API *pre-send filter 3* is actually *pre-send filter 4* to a designer.

Setting a Distance Model for a Group

You can set a distance model for a Group. Group distance model settings are inherited by all Group-assigned Sounds upon which the Distance Model property in Bank contents is set to "By Group". To set a Group's distance model you use the `sceScreamSetGroupDistanceModel()` function. You specify a target Group, and a hash of the name of a distance model, defined in the loaded distance model file; see Loading a Distance Model File. You can use the `sceScreamGetHashFromName()` function to obtain a hash from a distance model name.

Consultation Point: Your audio designer can supply you with a list of the names of distance models contained in a distance model file.

Designers can also set Group distance models in Scream Tool, and export these settings as part of a group mixer file; see Loading a Group Mixer File. When setting Group distance models using the `sceScreamSetGroupDistanceModel()` function, check with your audio designer to make sure you are not unintentionally overwriting their settings.

You can assign no distance model to a Group by setting the `sceScreamSetGroupDistanceModel()` function's *`modelNameHash`* parameter to 0; which is equivalent to a designer setting of "2D (none)". Sounds inheriting this setting do not attenuate based on distance.

Aligning Game-World and Scream Distance Units

Audio designers specify the range within which Sounds respond to distance model attenuation using Inner and Outer Radius properties. These properties, set on Sounds (or overridden on asset Grains), are specified in units of meters. Values specified for Sound and listener 3D positional coordinates are also assumed to be in meters. In your game, if distance is expressed in units other than meters, you can use the `sceScreamSetWorldUnitsPerMeter()` function to instruct Scream to convert your distance units to meters. Thereafter, you can specify Sound and listener positions in your game-world units, and Scream adjusts its distance calculations accordingly. You can also retrieve the current number of game-world units per meter using the `sceScreamGetWorldUnitsPerMeter()` function.

Retrieving 3D Parameter Data from Sounds

See 3D Designer Parameters and 3D Runtime Components in the "Working with Sounds" chapter.

Doppler Pitch Shifting

Doppler is the name given to the phenomenon of pitch shifting that occurs with a moving sound source or a moving listener. A common example would be an ambulance or fire truck siren that moves towards and then away from a stationery listener. The pitch of the sound source appears to rise as the sound becomes closer, and to fall as the sound again recedes into the distance.

There are two methods by which you can achieve Doppler pitch shifting in Scream:

- Use Scream's internal Doppler effect
- Implement custom Doppler pitch shifting using the `sceScreamGetDopplerPitchTranspose()` function

Note: Development teams using the `sceScreamGetDopplerPitchTranspose()` function for Doppler processing may wish to switch to the internal Doppler effect. However, Scream's internal Doppler effect uses essentially the same public APIs to calculate pitch shift amounts, so there is no additional *magic*. The primary advantage of Scream's internal Doppler effect is ease-of-use.

Working with Scream's Internal Doppler Effect

Scream provides a built-in Doppler effect, together with a number of APIs by which you exert control over it:

- A Sound-specific Doppler Factor property, with separate run-time and design-time factors
- A system-wide Doppler slew rate
- A Sound flag to manage camera cuts that might otherwise produce velocity discontinuities

Speed calculations are based on successive updates of a Sound's 3D positional coordinates with respect to those of a referenced listener, and the time deltas between updates. You set 3D coordinates on a Sound using the `SceScreamSoundParams` structure's *`position`* member. And you reference a listener using the `SceScreamSoundParams` *`listenerHandle`* member. A listener, in this context, is a proxy for a listening point in game world space. For example, this might be a game character or the location of a camera.

You create listeners using the `sceScreamCreateListener()` function. Scream supports up to `SCE_SCREAM_SND_MAXLISTENERS` (16) simultaneous listeners. Having created a listener you must set and dynamically update its position using the `sceScreamSetListener()` function. You specify the handle of a listener (returned by the `sceScreamCreateListener()` function) upon which to set coordinates in the `sceScreamSetListener()` function's *`handle`* parameter. And you specify the listener's position in game-world space in the *`location`* parameter, using a `SceScreamSnd3DVector` structure.

Note: For Doppler effect processing only the location of a listener is required, not the listener's orientation. Consequently, the `sceScreamSetListener()` function's *`front`* and *`top`* parameters are ignored.

Based on these values for Sound and listener position, and with respect to time intervals between successive updates, Scream automatically calculates velocities and applies the corresponding pitch shift to Sounds.

See the Scream Doppler example program for an illustration of Scream's internal Doppler effect in use.

Setting Doppler Factors

In addition to setting Sound and listener coordinates, you can exaggerate, diminish, or switch off Scream's Doppler effect by setting a Sound's Doppler Factor property. Often, you may need to exaggerate the effect for more dramatic results. Designers can set a Doppler Factor property on Sounds as part of Bank contents. Programmers can scale this value using the `SceScreamSoundParams` structure's *`dopplerFactor`* member. To switch off the Doppler effect you can set the *`dopplerFactor`* member to 0. To leave the Doppler Factor setting as per Bank contents set the *`dopplerFactor`* member to 1.0. An overall Doppler Factor of 1.0 produces a natural Doppler effect. To exaggerate the Doppler effect, set the *`dopplerFactor`* member to a value greater-than 1.0.

Setting the Doppler Slew Rate

The system-wide Doppler slew rate constrains the rate of change for Doppler velocity calculations, specifying the maximum allowable velocity change per second. Values are expressed in meters-per-second. You set the Doppler slew rate at initialization time using the `SceScreamPlatformInitEx2` *`dopplerSlewRate`* member. The *`dopplerSlewRate`* member defaults to `SCE_SCREAM_SND_DEFAULT_DOPPLER_SLEW_RATE` (that is, 240 meters-per-second).

Setting the …CAMERA_CUT Sound Flag

In a game scenario where the camera cuts to viewing from a different angle or location, and the camera location is associated with a referenced listener, the sudden change of location could produce a discontinuity in Scream's velocity calculations. To resolve this issue you can apply the flag `SCE_SCREAM_SND_FLAG_DOPPLER_CAMERA_CUT` to the `SceScreamSoundParams` structure's *`flags`* member. This informs the Doppler calculation mechanism that a location discontinuity is occurring this frame so that a large instantaneous Doppler shift is not attempted.

Aligning Game-World and Scream Distance Units

Values specified for Sound and listener 3D positional coordinates are assumed to be in meters. In your game, if distance is expressed in units other than meters, you can use the `sceScreamSetWorldUnitsPerMeter()` function to instruct Scream to convert your distance units to meters. Thereafter, you can specify Sound and listener positions in your game-world units, and Scream adjusts its Doppler velocity calculations accordingly. You can also retrieve the current number of game-world units per meter using the `sceScreamGetWorldUnitsPerMeter()` function.

Implementing Custom Doppler Pitch Shifting

To implement custom Doppler pitch shifting in Scream, you can use the `sceScreamGetDopplerPitchTranspose()` function to calculate pitch transposition amounts, and apply the output to the `SceScreamSoundParams` structure's *`pitchTranspose`* member. The `sceScreamGetDopplerPitchTranspose()` function calculates a pitch transposition amount based on an input approaching speed parameter, expressed in meters-per-second.

The `sceScreamGetDopplerPitchTranspose()` function returns a pitch transposition amount expressed in fines (128th microtonal subdivisions of a semitone), which you can apply directly as input to the `SceScreamSoundParams` structure's *`pitchTranspose`* member. You calculate the speed (in meters-per-second) of the sound-emitter as it approaches the listener and then recedes into the distance. When the sound-emitter is moving away from the listener, pass a negative value for the *`approachingMps`* parameter.

Note: If using a custom Doppler solution, you should ensure that Scream's internal Doppler effect is switched off by setting the `SceScreamSoundParams` *`dopplerFactor`* member to 0 for all Sounds.

Game world distance may be measured in units other than meters, which may require some conversion for calculating speeds in meters-per-second.

Doppler pitch shift can sometimes be less than impressive when simulated in a game. You may need to exaggerate the effect for more dramatic results. One way to exaggerate it is to add a small multiplier to the calculated speed of an object or listener. We recommend that you implement this multiplier as a variable, and make it accessible from a debug menu, so that you can experiment with the value at run time.

Doppler Pitch Shift in a Multi-Player Game Context

Using Doppler pitch shift in multi-player, split-screen games can rapidly become complicated. Some development teams chose to turn off Doppler pitch shift effects in multi-player modes.

Sound Spatialization

Note: Many applications calculate three-dimensional coordinates of objects relative to the camera using more optimized methods. The utility functions outlined below, are intended for game scenarios in which no other spatialization data is available.

Using a set of utility functions, you can calculate the polar coordinates of a sound-emitting object relative to a listener, based on game world Cartesian coordinates. A *listener*, in this context, is a proxy for a listening point in game world space. For example, this might be a game character's ears, or in a sports game, the location of a camera. Calculating polar coordinates for a sound-emitting object relative to a listener involves the following tasks:

(1) Create a sound spatialization listener using the `sceScreamCreateListener()` function. Scream provides for up to `SCE_SCREAM_SND_MAXLISTENERS` (16) simultaneous listeners.

(2) Set the location and orientation of the listener using the `sceScreamSetListener()` function. As well as the handle of the listener you are setting, you specify the location, and the front and top orientation vectors of the listener using the `SceScreamSnd3DVector` data structure.

(3) Use the `sceScreamCalcSoundAngles()` function to calculate the azimuth and elevation angles of a sound-emitting object relative to a listener. You specify the location of the sound-emitting object using the `SceScreamSnd3DVector` data structure.

The `sceScreamCalcSoundAngles()` function saves its output azimuth and elevation values into user-supplied variables. The azimuth value is expressed in degrees: 0 is directly in front of and 180 is directly behind the listener. You can apply the azimuth value directly to the `SceScreamSoundParams` *`azimuth`* member in respect of the Sound that represents the sound-emitting object. The elevation value is also expressed in degrees: 0 is at listener height, 90 directly above, and -90 directly below the listener. The `SceScreamSoundParams` structure does not include an *`elevation`* member. Instead, Scream provides a way to manage distance cues through panning focus. Closer sounds, which tend to be more dispersed, are given wider focus. Conversely, more distant sounds are given narrower focus. Developers are free to choose from a variety of potential approaches for mapping elevation to focus, or even to ignore this value. One approach, for example, could be a simple linear calculation such as:

```
focus = ( abs(elevation) / 90 ) * 360 )
```

The following code sample shows how to set up a listener and calculate polar coordinates for a sound-emitting object, in this case, an alarm sound.

```
// declare structure for sound-emitting object (alarm sound)
SceScreamSoundParams alarm_params;

// declare structure for location of alarm
SceScreamSnd3DVector alarm_location;

// declare structures for listener location and orientation
SceScreamSnd3DVector player_location;
SceScreamSnd3DVector player_front;
SceScreamSnd3DVector player_top;

// player location
player_location.fX = 5.01;
player_location.fY = 3.25;
player_location.fZ = -2.5;

// player front orientation
player_front.fX = 0.1;
player_front.fY = 10.0;
player_front.fZ = 0.5;

// player top orientation
player_top.fX = 0.5;
player_top.fY = 0.1;
player_top.fZ = 10.0;

// alarm location
alarm_location.fX = 41.3;
alarm_location.fY = 18.75;
alarm_location.fZ = 10.25;

// create player listener
player_listener = sceScreamCreateListener();

// set location, and top and front oritentaion vectors
sceScreamSetListener(player_listener, &player_location, &player_front,
&player_top);

// calculate azimuth/elevation angles with respect to location of alarm
sceScreamCalcSoundAngles(player_listener, &alarm_location, &alarm_az,
&alarm_el);

// set alarm sound azimuth and focus
alarm_params.mask = SCE_SCREAM_SND_MASK_PAN_AZIMUTH |
SCE_SCREAM_SND_MASK_PAN_FOCUS;
alarm_params.azimuth = alarm_az;

// simple approach to mapping elevation to focus
// (higher/lower the elevation, wider the focus...)
alarm_params.focus = ( abs(alarm_el) / 90 ) * 360 );
```

Understanding Reverberation

This chapter provides a conceptual understanding of reverb, both generally and in a game audio context.

What is Reverberation?

Reverberation (or reverb) is the persistence of sound in an indoor or outdoor acoustic environment after the sound source has stopped. Sound, in this context is defined as air pressure waves that emanate outwards from a source in all directions. You can visualize sound propagating through the medium of air as a stream of spherical ripples. If the environment in which a sound is projected has reflective surfaces (walls, ceiling, floor, and so on), the resultant air pressure waves continue to reflect off the surfaces until their energy is eventually absorbed, and they are no longer audible.

Stages of Reverberation

From the onset of the direct signal until the last of the reflections decay, three distinct stages of reverberation can be identified.

Direct Signal

Direct signal is the perception of air pressure waves directly from the sound source, without having reflected off any intermediary surfaces. There can only be one direct path (trajectory) between the sound source and the listener, and, because it is the shortest distance between these two points, the direct signal reaches the listener first. Direct signal is known as dry signal: it is the original sound source without any reverb effect added.

Early Reflections

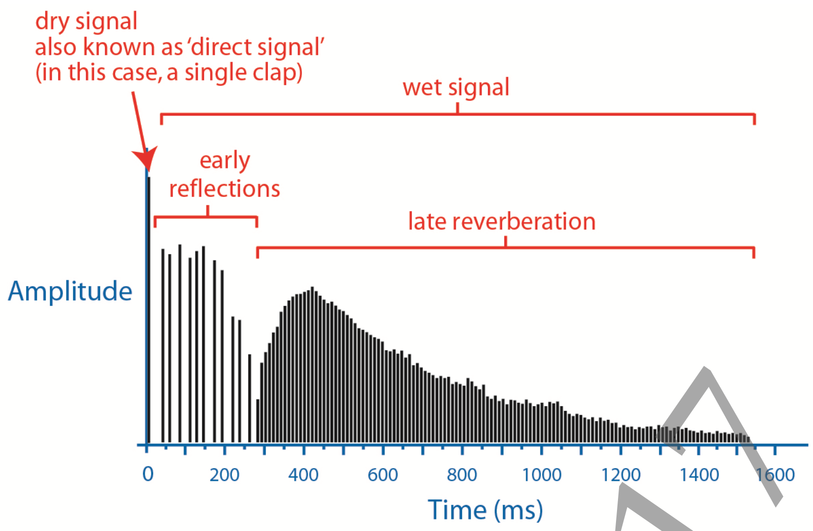
Of the infinite number of trajectories that air pressure waves can follow in an enclosed space, a relatively small number reach the listener having reflected off only one or two surfaces. These early reflections reach the listener after the direct signal, but before the onset of late reverberation. Early reflections are heard as a short series of discrete echoes.

Late Reverberation

As time progresses and the mass of air pressure waves have reflected off numerous surfaces, a general ambience around the sound source starts to build up. These diffuse reflections are so closely packed and so numerous that they cannot be perceived discretely. The time taken for late reverberation to build up and decay is a significant attribute. It depends on the dimensions of the environment and the reflectivity of its surfaces. The combination of early reflections and late reverberation is known as wet signal.

Figure 17 illustrates the stages of reverberation: a short burst of sound (a single clap) is heard in a mid-size hall:

Figure 17 Stages of Reverberation: A Single Clap is Heard in a Mid-size Hall



Reverberation Time and Decay

Reverberation generally takes time to build up and decay. This is known as reverberation time (or by the more technical term, RT60, the time taken for the reverb to decay by 60 dB from its peak energy). Reverberation time is dependent on the reflectivity of the environment in which the sound is heard. Hard, flat surfaces, such as those made of stone, glass, or ceramic tile, can be very reflective, producing longer reverberation times. Soft, uneven surfaces, such as those made of fabric, tend to absorb air pressure waves more quickly, producing shorter reverberation times.

Different frequency components within a reverberated sound do not decay uniformly. Higher frequencies have shorter wavelengths and usually less energy (due to smaller amplitude) than the lower frequencies. This means that high frequencies tend to be absorbed more readily by the surrounding surfaces and by the air itself. Reverberation therefore, tends to behave like a low-pass filter, in which higher frequencies are attenuated.

Spectral Character of Late Reverberation

You can think of the spectral character of late reverberation as colored noise; it results from the acoustic properties of the space in which the sound is heard. Essentially, the space itself, its dimensions, geometry, and the nature of its surfaces, behaves as a large resonating chamber, similar, for example, to the body of an acoustic guitar. The wavelengths of some frequency components align with the dimensions of the space, and resonate more freely. Wavelengths of other frequency components may be out of alignment with room dimensions, and thus attenuated. Ultimately, you can think of reverberation as a diffuse resonator with spectral characteristics resulting from the acoustic properties of the surrounding environment together with the discrete echo properties of the early reflections.

Dry and Wet Signals

Two commonly used terms in reverb simulation are dry and wet signals. Dry signal is the direct signal (that is, the original waveform), untreated by reverb. The wet signal is a processed version of the direct signal that has been treated with a reverb effect. The dry and wet signals are mixed together to simulate the sound of the original waveform heard in some environment.

The balance between dry and wet signals is important to the overall effect. It provides a way to simulate the distance between the sound source and the listener. If the listener is close to the sound source, the dry signal tends to be louder than the wet signal. Conversely, if the listener is far from the sound source, the wet signal tends to be louder than the dry signal.

Reverb in a Game Audio Context

The preceding discussion focused on general concepts, and assumed a music-oriented model in which sound source and listener are both fixed and in the same indoor environment. Reverb in a game audio context can make no such assumptions:

- sound source and listener may not be in the same environment
- there may be obstacles in the environment(s) requiring additional audio rendering
- sound source, listener, or both may be in motion

In the game context, reverb is part of audio rendering. It tries to physically model sound propagation within the virtual space(s) featured in the game. In the music-oriented model, a single reverb setting generally suffices for an entire song. In game audio, a potentially large number of reverb settings might be required, depending on the number of virtual spaces featured in the game. These settings can be used for only short durations, specific to a particular level or stage of the game. In some cases, multiple simultaneous reverb settings may be required (for example, in games supporting multiple remote players). Reverb settings often have interactive dependencies, determined at run time by the game engine and user responses.

Reverb in a game audio context is dynamic and interactive; in a typical musical application it is static and passive. The requirements of reverb for game audio are amply supported by the I3DL2 reverb, which enables game audio designers and programmers to work together to produce convincing simulations of the environment(s) featured in a game.

Tutorial

This tutorial provides an introduction to programming game audio content in the Scream Runtime. In it, you learn how to initialize and start the Scream Runtime (and its dependencies), how to load a Bank file, how to play Sounds (contained in the loaded Bank), and how to terminate the Scream Runtime. The Bank file used in this tutorial is included with the Scream distribution. A companion Scream Tool tutorial is available in the *Scream Tool Help*. It explains how to create the Sounds and export the Bank file used in this tutorial.

Topics covered in this tutorial:

Required SDK Library, Header, and Asset Files

The example application you create in this tutorial requires various library, header, and asset files.

Library and Header Files

SDK header and library files required for this tutorial application are listed in Table 32.

Table 32 Required SDK Header and Library Files

File	Description
System Headers	
`<string.h>`	Defines functions to manipulate strings, arrays, and memory.
`<stdio.h>`	Defines functions to perform input/output operations.
`<stdlib.h>`	Defines some general purpose functions.
`<kernel.h>`	Defines general system entry points.
`<kernel/types.h>`	Defines general system types.
`<kernel/threadmgr.h>`	Supports system threading.
`<libsysmodule.h>`	Header for system module handling, used to initialize the audio hardware.
Scream Headers	
`<scream/sce_scream.h>`	Prototypes for the Scream public interface and definitions of Scream data structures.
NGS Libraries	
`libSceSysmodule_stub.a`	Library for system module handling, used to initialize the audio hardware.
`libSceNgs_stub_weak.a`	The NGS synthesizer library file.
`libSceAudio_stub.a`	The lowest-level audio hardware library, used by the NGS synthesizer.
`libSceScream.a`	The main Scream library file.
NGS2 Libraries	
`libSceNgs2_stub_weak.a`	The NGS2 synthesizer library file.
`libSceAudioOut_stub.a`	The lowest-level audio hardware library, used by the NGS2 synthesizer.
`libSceScream.a`	The main Scream library file.

Note: The FIOS header and library files are required for applications that use streaming functionality provided by Sndstream. However, the simple application outlined in this tutorial employs memory-resident assets only. Therefore, `fios2.h` and `libSceFios2_stub.a` are not included.

System Declarations

In addition to including the above header and library files, before initializing the Scream Runtime component, you must provide the following system declarations.

```
// System Declarations
const char sceUserMainThreadName[] = "audio_example_main_thread";
int sceUserMainThreadPriority = SCE_KERNEL_DEFAULT_PRIORITY_USER;
unsigned int sceLibcHeapSize = 16*1024*1024; // 16 MB
```

Bank Asset File

The binary Bank file used in this tutorial is included with the Scream distribution. You can also create the file by doing the Scream Tool tutorial; see the "Tutorial" chapter of the *Scream Tool Help*. With respect to platform SDK, the binary Bank file, `SwordFight.bnk`, installs in the following directories:

- `%SCE_PSP2_SDK_DIR%/data/audio_video/sound/wave/`
- `%SCE_ORBIS_SDK_DIR%/data/audio_video/sound/wave/`

Initializing and Starting the Scream Runtime and its Dependencies

The first task is to initialize and start the Scream Runtime component. You start the Scream Runtime with the `sceScreamStartSoundSystemEx2()` function, specifying as the function's only argument a `SceScreamPlatformInitEx2` structure appropriately initialized for your game. Before initializing a `SceScreamPlatformInitEx2` structure, however, you must load the NGS module, and implement the required custom memory callback functions.

Initialize the Platform

Any platform-specific initialization tasks must be done prior to initializing Scream.

PlayStation®Vita

On the PlayStation®Vita platform, applications must first load the NGS module before initializing the Scream Runtime.

You use the `sceSysmoduleLoadModule()` function to load the NGS module, specifying (as the function's only argument) the constant associated with the NGS module, `SCE_SYSMODULE_NGS`. Optionally, you can add a `printf` statement to report any resulting errors.

```
// Load the NGS module
int result = sceSysmoduleLoadModule( SCE_SYSMODULE_NGS );
if( result != SCE_OK )
 {
  printf( "initNGS: sceSysmoduleLoadModule() failed: 0x%08X\n", result );
  return -1;
 }
```

PlayStation®4

On the PlayStation®4 platform, applications must first load the NGS2 module before initializing the Scream Runtime.

You use the `sceSysmoduleLoadModule()` function to load the NGS2 module, specifying (as the function's only argument) the constant associated with the NGS2 module, `SCE_SYSMODULE_NGS2`. Optionally, you can add a `printf` statement to report any resulting errors.

```
// Load the NGS2 module
int result = sceSysmoduleLoadModule( SCE_SYSMODULE_NGS2 );
if( result != SCE_OK )
 {
  printf( "initNGS2: sceSysmoduleLoadModule() failed: 0x%08X\n", result );
  return -1;
 }
```

Implement Custom Memory Callback Functions

Having loaded the NGS or NGS2 modules, you can implement the required custom memory allocation and memory free callback functions, prototypes for which are shown below.

```
typedef void * (*SceScreamExternSndMemAlloc)(int32_t bytes, int32_t use);
typedef void (*SceScreamExternSndMemFree)(void *mem);
```

Scream invokes these callback functions whenever memory allocation or memory release is required. An example implementation follows.

```
// Memory allocation callback function
void* example_malloc(int32_t bytes, int32_t use)
{
 void* p;

  p = malloc(bytes);
  if (p) return p;
  return NULL;
}

// Memory release callback function
void example_free(void *memory)
{
    free(memory);
}
```

Initialize a SceScreamPlatformInitEx2 Structure

Next, you initialize a `SceScreamPlatformInitEx2` structure, setting any members that require application-specific values. In this tutorial, we set only the following members:

- *`memAlloc`* –Address of an application-defined memory allocation function.
- *`memFree`* – Address of an application-defined memory release function.

After instantiating a `SceScreamPlatformInitEx2` structure, you can set default values for its members using the `sceScreamFillDefaultScreamPlatformInitArgsEx2()` function as follows:

```
// Instantiate SceScreamPlatformInitEx2 and set default values
SceScreamPlatformInitEx2 screamInit;
memset(&screamInit, 0, sizeof(SceScreamPlatformInitEx2));
screamInit.size = sizeof(SceScreamPlatformInitEx2);
sceScreamFillDefaultScreamPlatformInitArgsEx2(&screamInit);
```

Now you can set application-specific values for the members listed above.

```
screamInit.memAlloc = example_malloc;
screamInit.memFree = example_free;
```

Start the Scream Runtime

Finally, you can call the `sceScreamStartSoundSystemEx2()` function, specifying a pointer to your initialized `SceScreamPlatformInitEx2` structure. Optionally, you can add a `printf` statement to report any resulting errors.

```
// Start the Scream Runtime
int32_t result;
result = sceScreamStartSoundSystemEx2(&screamInit);
if( result != 0 )
 {
   printf("sceScreamStartSoundSystemEx2 failed, returned %d\n", result );
 }
```

Loading a Bank File

Audio designers export Sounds (consisting of audio assets and associated playback scripts) in the form of binary Bank files (`BNK`), which you load into the Scream Runtime.

The Scream API provides two functions for loading binary Bank files. You can either load a `BNK` file from memory by calling the `sceScreamBankLoadFromMemEx()` function , or you can load a `BNK` file from disk by calling the `sceScreamBankLoadEx()` function. The latter approach is described in this tutorial.

Load a Bank File from Disk

You load a binary Bank file from disk using the `sceScreamBankLoadEx()` function, specifying the file path and offset of a `BNK` file to load. The *`offset`* parameter provides a way to reference the start of a `BNK` file that may be embedded in a larger container file. If the load is successful, the function returns a `SceScreamSFXBlock2` pointer to the loaded Bank.

```
// Load Bank binary
#define BANK_NAME "app0:/audio_video/sound/wave/SwordFight.bnk"  // PlayStation®Vita path
#define BANK_NAME "app0/audio_video/sound/wave/SwordFight.bnk"   // PlayStation®4 path
SceScreamSFXBlock2 *screamBank = sceScreamBankLoadEx( (const char*) BANK_NAME, 0);
```

Playing Sounds

After loading a Bank, you can set Sound parameters and play Sounds contained in the Bank.

Initialize a SceScreamSoundParams Structure

First, prepare a `SceScreamSoundParams` structure. The `SceScreamSoundParams` structure stores Sound parameter values that can be used for playback. In addition to setting the *`size`* member, which is required, we set the structure's *`gain`* and *`azimuth`* members. You must also indicate which parameters contain settings, by including their corresponding flags in the structure's *`mask`* member as follows.

```
// Instantiate SceScreamSoundParams structure and set members
SceScreamSoundParams soundParams;
memset(&soundParams, 0, sizeof(SceScreamSoundParams));
soundParams.size = sizeof(SceScreamSoundParams);
soundParams.gain = 1.0f;
soundParams.azimuth  = 0;
// Set flags to indicate which parameters are being set
soundParams.mask = SCE_SCREAM_SND_MASK_GAIN|SCE_SCREAM_SND_MASK_PAN_AZIMUTH;
```

Play Sounds

You can play Sounds by addressing them either by name (using the `sceScreamPlaySoundByNameEx()` function) or by index (using the `sceScreamPlaySoundByIndexEx()` function). In this tutorial we address Sounds by name. You specify the Bank handle of the loaded `BNK` file returned by the `sceScreamBankLoadEx()` function, the name of a Sound you wish to play, and a pointer to the `SceScreamSoundParams` structure containing your gain and azimuth parameter settings. To separate playback of the three Sounds contained in the Bank, we add a 3-second delay between each Sound.

```
// Play Sounds contained in the SwordFight.bnk file
// with a 3-second delay between each Sound
sceScreamPlaySoundByNameEx(screamBank, "SwordFight-1", &soundParams);
sceKernelDelayThread(3000000);        // PlayStation®Vita
sceKernelSleep(3);                    // PlayStation®4

sceScreamPlaySoundByNameEx(screamBank, "SwordFight-2", &soundParams);
sceKernelDelayThread(3000000);        // PlayStation®Vita
sceKernelSleep(3);                    // PlayStation®4

sceScreamPlaySoundByNameEx(screamBank, "SwordFight-3", &soundParams);
```

Terminating Processing and Stopping the Scream Runtime

The procedure to terminate audio processing and stop the Scream Runtime is described as follows.

Stop the Scream Runtime

You shut down the Scream Runtime by calling the `sceScreamStopSoundSystem()` function. The functions stops all active synthesizer voices and releases the Runtime's allocated memory. Once the Scream Runtime has been stopped, you can unload the NGS or NGS2 module by calling the `sceSysmoduleUnloadModule()` function.

```
// Stop the Scream Runtime
sceScreamStopSoundSystem();
// Unload the NGS or NGS2 modules
sceSysmoduleUnloadModule( SCE_SYSMODULE_NGS );       // PlayStation®Vita
sceSysmoduleUnloadModule( SCE_SYSMODULE_NGS2 );      // PlayStation®4
```

Complete Example Code

Complete example code used in this tutorial, including all relevant library headers and modules, is shown below. Due to minor differences between platforms, the PlayStation®Vita and PlayStation®4 example code appear separately.

PlayStation®Vita

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include <kernel.h>
#include <kernel/types.h>
#include <kernel/threadmgr.h>

#include <libsysmodule.h>

#include <scream/sce_scream.h>

#define BANK_NAME "app0:/audio_video/sound/wave/SwordFight.bnk"

// System Declarations
const char sceUserMainThreadName[]        = "audio_example_main_thread";
int sceUserMainThreadPriority = SCE_KERNEL_DEFAULT_PRIORITY_USER;
unsigned int sceLibcHeapSize = 16*1024*1024; // 16 MB

// Memory Handling
void* example_malloc(int32_t bytes, int32_t use)
{
    void* p;

    p = malloc(bytes);
    if (p) return p;
    return NULL;
}

void example_free(void *memory)
{
    free(memory);
}

int main()
{
   // Load the NGS Module
   int result = sceSysmoduleLoadModule( SCE_SYSMODULE_NGS );
   if( result != SCE_OK )
   {
     printf( "initNGS: sceSysmoduleLoadModule() failed: 0x%08X\n", result );
     return -1;
   }

   // Initialize Scream
   SceScreamPlatformInitEx2 screamInit;
   memset(&screamInit, 0, sizeof(SceScreamPlatformInitEx2));
   screamInit.size = sizeof(SceScreamPlatformInitEx2);
   sceScreamFillDefaultScreamPlatformInitArgsEx2(&screamInit);
   screamInit.memAlloc        = example_malloc;
   screamInit.memFree         = example_free;

   result = sceScreamStartSoundSystemEx2(&screamInit);
   if( result != 0 )
   {
     printf("sceScreamStartSoundSystemEx2 failed, returned %d\n", result );
   }

   // Load the tutorial Bank (BNK) file
   SceScreamSFXBlock2 *screamBank = sceScreamBankLoadEx( (const char*) BANK_NAME, 0 );
   if ( screamBank == NULL)
   {
     printf("sceScreamBankLoadEx failed.\n" );
   }

    // Initialize a SceScreamSoundParams structure
    SceScreamSoundParams soundParams;
    memset(&soundParams, 0, sizeof(SceScreamSoundParams));
    soundParams.size      = sizeof(SceScreamSoundParams);
    soundParams.gain     = 1.0f;
    soundParams.azimuth   = 0;
    soundParams.mask = SCE_SCREAM_SND_MASK_GAIN|SCE_SCREAM_SND_MASK_PAN_AZIMUTH;

    // Play Sounds contained in the SwordFight.bnk file
    // with a 3-second delay between each Sound
    sceScreamPlaySoundByNameEx(screamBank, "SwordFight-1", &soundParams);
    sceKernelDelayThread(3000000);

    sceScreamPlaySoundByNameEx(screamBank, "SwordFight-2", &soundParams);
    sceKernelDelayThread(3000000);

    sceScreamPlaySoundByNameEx(screamBank, "SwordFight-3", &soundParams);

    // Terminate processing and unload the NGS module
    sceScreamStopSoundSystem();
    sceSysmoduleUnloadModule( SCE_SYSMODULE_NGS );

    return 0;
}
```

PlayStation®4

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include <kernel.h>
#include <kernel/types.h>
#include <kernel/threadmgr.h>


#include <scream/sce_scream.h>

#define BANK_NAME "/app0/audio_video/sound/wave/SwordFight.bnk"

// System Declarations
const char sceUserMainThreadName[]        = "audio_example_main_thread";
int sceUserMainThreadPriority = SCE_KERNEL_DEFAULT_PRIORITY_USER;
unsigned int sceLibcHeapSize = 16*1024*1024; // 16 MB


// Memory Handling
void* example_malloc(int32_t bytes, int32_t use)
{
    void* p;

    p = malloc(bytes);
    if (p) return p;
    return NULL;
}

void example_free(void *memory)
{
    free(memory);
}

int main()
{
   // Load the NGS2 Module
   int result = sceSysmoduleLoadModule( SCE_SYSMODULE_NGS2 );
   if( result != SCE_OK )
   {
     printf( "initNGS2: sceSysmoduleLoadModule() failed: 0x%08X\n", result );
     return -1;
   }

   // Initialize Scream
   SceScreamPlatformInitEx2 screamInit;
   memset(&screamInit, 0, sizeof(SceScreamPlatformInitEx2));
   screamInit.size = sizeof(SceScreamPlatformInitEx2);
   sceScreamFillDefaultScreamPlatformInitArgsEx2(&screamInit);
   screamInit.memAlloc         = example_malloc;
   screamInit.memFree          = example_free;

   result = sceScreamStartSoundSystemEx2(&screamInit);
   if( result != 0 )
   {
     printf("sceScreamStartSoundSystemEx2 failed, returned %d\n", result );
   }

   // Load the tutorial Bank (BNK) file
   SceScreamSFXBlock2 *screamBank = sceScreamBankLoadEx( (const char*) BANK_NAME, 0 );
   if ( screamBank == NULL)
   {
     printf("sceScreamBankLoadEx failed.\n" );
   }
```

```
    // Initialize a SceScreamSoundParams structure
    SceScreamSoundParams soundParams;
    memset(&soundParams, 0, sizeof(SceScreamSoundParams));
    soundParams.size     = sizeof(SceScreamSoundParams);
    soundParams.gain    = 1.0f;
    soundParams.azimuth   = 0;
    soundParams.mask = SCE_SCREAM_SND_MASK_GAIN|SCE_SCREAM_SND_MASK_PAN_AZIMUTH;


    // Play Sounds contained in the SwordFight.bnk file
    // with a 3-second delay between each Sound
    sceScreamPlaySoundByNameEx(screamBank, "SwordFight-1", &soundParams);
    sceKernelSleep(3);

    sceScreamPlaySoundByNameEx(screamBank, "SwordFight-2", &soundParams);
    sceKernelSleep(3);

    sceScreamPlaySoundByNameEx(screamBank, "SwordFight-3", &soundParams);

    // Terminate processing and unload the NGS2 module
    sceScreamStopSoundSystem();
    sceSysmoduleUnloadModule( SCE_SYSMODULE_NGS2 );

    return 0;
}
```