Sndstream Library Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document

This document is a detailed guide to using the Sndstream 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

Sndstream is a library and runtime component for playing audio files on the PlayStation®Vita and PlayStation®4 platforms. This document is a guide to programming the Sndstream runtime library, and is a counterpart to the *Sndstream Library Reference*.

Sndstream manages tasks such as reading audio files from disk, decoding compressed audio data, and filling audio buffers for playback. Sndstream operates as a plug-in to Scream. Sndstream requires Scream and the NGS or NGS2 synthesizers for audio playback, allocating synthesizer voices for each channel of played audio, and assigning Streams to Scream volume groups. The handles returned by the Stream-starting functions (`sceScreamStartStream()`, and `sceScreamStartStreamByFileToken()`) can also be used in Scream functions calls. This allows them to be manipulated along with other Scream Sounds.

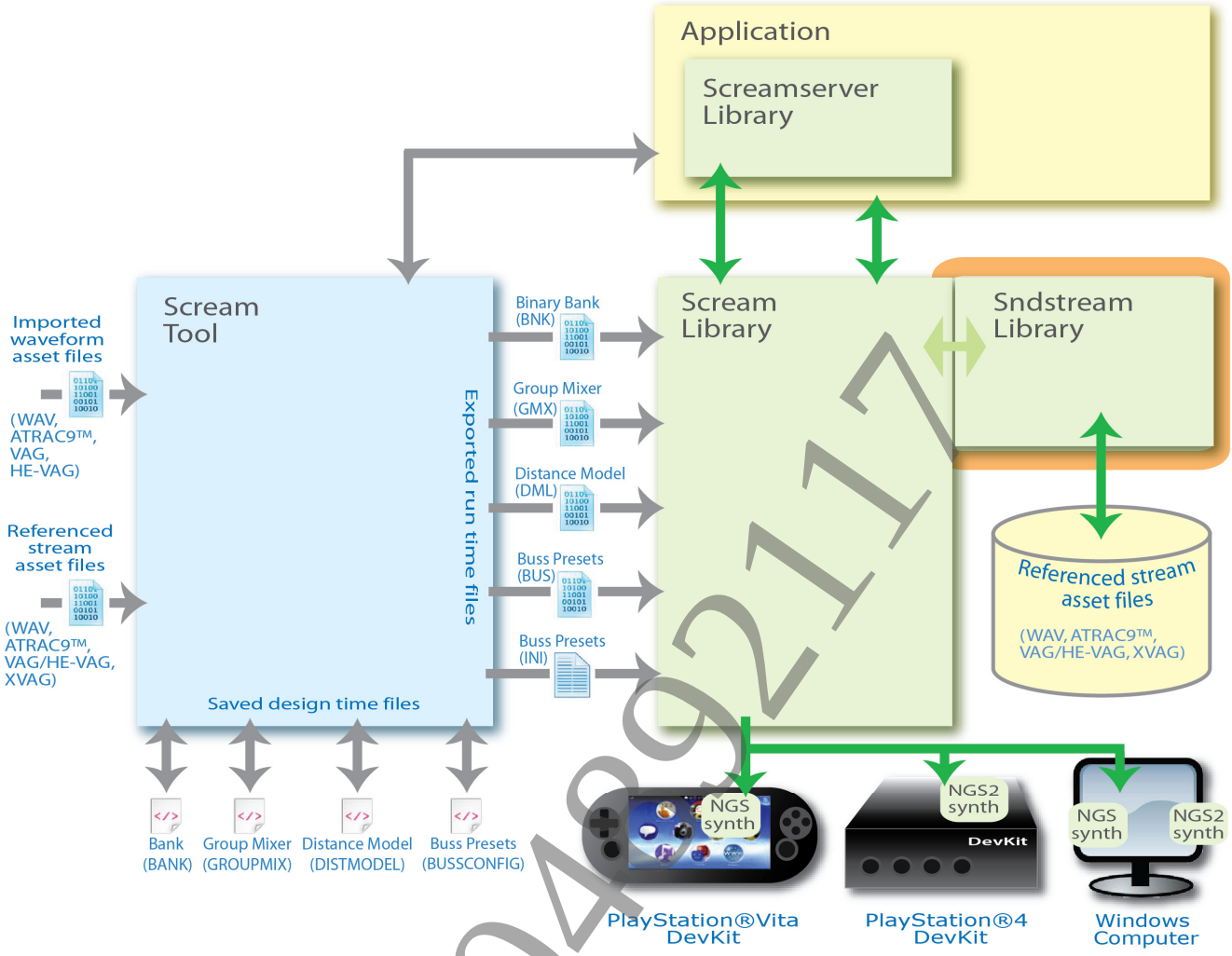
In Sndstream, all data reading is performed asynchronously from the Scream tick. Sndstream supports both synchronous and asynchronous custom file I/O functions. See Using Custom File I/O Functions in the "Working with System Globals" chapter for details.

Sndstream supports game interactive control over audio content in a number of ways.

- Music tracks can smoothly transition from one Stream to the next in accordance with game contexts, and on predetermined musical boundaries (see the Synchronizing Stream Transitions, Overlays, and Scream Sounds chapter).
- You can stitch together dialog fragments to form game-dependent speech by manipulating the list of items queued to a Stream (see Inserting a File into the Queue of an Existing Stream in the "Manipulating an Active Stream" chapter).
- You can manipulate individual Layers within a multi-Layer Stream file, as if live-mixing a multi-track recording (see the Working with Multi-Layer Streams chapter for details).

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

Figure 1 Scream Audio System Topology



Features

Sndstream offers the following features:

- Plays multiple simultaneously streamed audio files asynchronously from the Scream tick, requiring minimal system resources.
- Supports multiple compressed and uncompressed file formats, including WAV, VAG, ATRAC9™, in mono and stereo.
- Supports multi-channel audio file streaming with independent channel volume control
- Can start playback of queued audio files immediately after another file finishes or simultaneously with other queued audio file(s).
- Transitions Streams smoothly on precise musical boundaries.
- Supports dynamic loop playback manipulation.
- Supports custom file I/O and memory allocation functions.
- Supports playback from a file image in RAM or from disk.
- Supports pre-parsing of audio file headers and an efficient Stream file referencing method by file token.

Platform Compatibility

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

Embedding into a Program

For details on embedding the Scream library into a program, see the section "Embedding into a Program" in the "Introduction" chapter of the *Scream Library Overview* document. The only additional requirements to embed the Sndstream library are described as follows.

Sndstream Library File

The Sndstream header file is included within the Scream header file `scream_top.h`. The library file required for using Sndstream on the PlayStation®Vita and PlayStation®4 platforms is listed in Table 1.

Table 1 Required Sndstream Library File

Required File	Description
libSceSndstream.a	The main Sndstream library. Includes all functionality described here.

FIOS

Applications must also initialize FIOS if not providing a custom file I/O interface.

Sample Programs

The PlayStation®Vita and PlayStation®4 SDKs include a number of Sndstream sample programs. With respect to platform SDK, 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_libsndstream/common/audio_examples.h`
- `sample_code/audio_video/api_libsndstream/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_libsndstream` 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_libsndstream.sln`
PlayStation®4	`api_libsndstream.sln`

MIDI Sync

Sndstream MIDI Sync demonstrates synchronized Stream transitions. The sample begins by playing *SONG_A*, then transitions to *SONG_B* on a precise musical boundary defined by the `SceScreamSndSyncParams` structure's *`syncUnit`* and *`unitMultiple`* members. For further details, see the Synchronizing Stream Transitions, Overlays, and Scream Sounds chapter.

PlayStation®Vita	`sample_code/audio_video/api_libsndstream/sndstream_midisync`
PlayStation®4	`sample_code/audio_video/api_libsndstream/midisync`

Play Info

Sndstream Play Info demonstrates use of the information retrieval functions. The sample retrieves and prints out buffered status, channel count, and sample rate for an active Stream using the `sceScreamGetStreamInfo()` function. It also retrieves and prints out the current playback location using the `sceScreamGetStreamFileLocationInSeconds()` function. For further details, see the Retrieving Stream Information chapter.

On the PlayStation®4 platform, the Sndstream Play Info sample also demonstrates manipulation of Stream Layers. For further details, see the Working with Multi-Layer Streams chapter.

PlayStation®Vita	`sample_code/audio_video/api_libsndstream/sndstream_playinfo`
PlayStation®4	`sample_code/audio_video/api_libsndstream/playinfo`

Stitch

Sndstream Stitch demonstrates sample-accurate stitching of successive Stream files. The sample makes use of the `sceScreamQueueToStream()` function to dynamically insert a file into the queue of an active Stream, producing seamless playback. For further details, see Inserting a File into the Queue of an Existing Stream in the "Manipulating an Active Stream" chapter.

PlayStation®Vita	`sample_code/audio_video/api_libsndstream/sndstream_stitch`
PlayStation®4	`sample_code/audio_video/api_libsndstream/stitch`

Preparse

Sndstream Preparse demonstrates creation and manipulation of file tokens and file token storage, and starting and queuing Streams by reference to a file token. For further details, see the Working with File Tokens chapter.

PlayStation®Vita	`sample_code/audio_video/api_libsndstream/sndstream_preparse`
PlayStation®4	`sample_code/audio_video/api_libsndstream/preparse`

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, and a guide document describing use of the BankMerge/Build utility. The following is a complete list of Scream library documents:

- *Scream Library Overview* — A guide to programming the Scream library.
- *Scream Library Reference* — A reference to the Scream library API, used in conjunction with the NGS2 synthesizer running on the PlayStation®4 platform.
- *Scream Library Reference* — A reference to the Scream library API, used in conjunction with the NGS synthesizer running on the PlayStation®Vita platform.
- *Sndstream Library Overview* (this document) — 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.

If you are not familiar with such terms as "NGS", see the *Scream Audio Glossary*. You should also be familiar with the Scream runtime to understand this document.

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 of the *Scream Library Overview*.

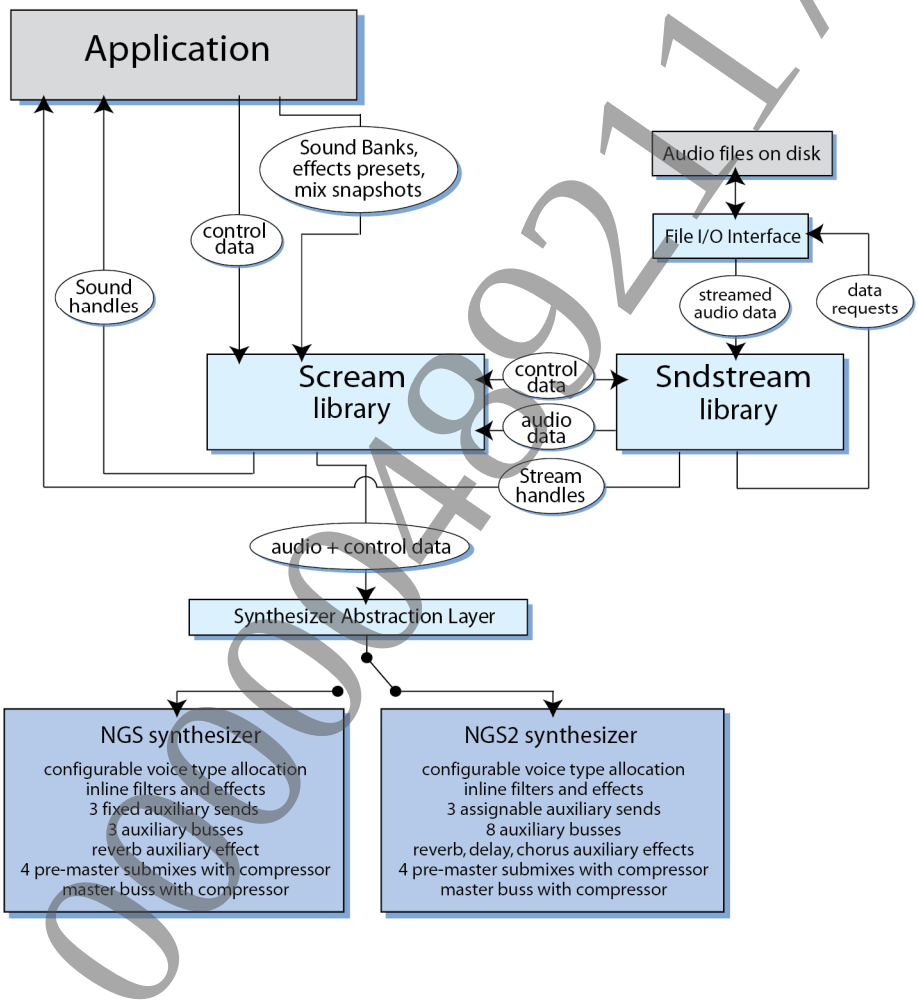
2 System Overview

The Sndstream runtime component operates as an add-on to the Scream runtime component. It extends Scream's functionality to provide audio file streaming support.

When you start a Stream in Sndstream, the system returns a Stream handle to the application, which can be used to reference the Stream in both Sndstream and Scream function calls. That is, you can reference the Stream handle to perform a wide range of runtime manipulation, both independently and in conjunction with other Scream Sounds (and Sndstream Streams).

Figure 2 depicts the flow of audio and control data between the various components. For a design time depiction of the components and data flow, see "Auditioning Servers" in the *Scream Tool Help*.

Figure 2 Runtime Depiction of Scream and Sndstream Audio and Control Data Flow



Memory Allocation

When Sndstream is initialized, memory requirements are allocated in one block. Internally, Sndstream allocates memory using Scream function calls, which provide a way to override Sndstream memory allocation using Scream's memory allocation override functionality. For further details, see the `SceScreamExternSndMemAlloc()` function in the "Scream Type Definitions" section of the *Scream Library Reference* documents.

Supported File Types

Table 2 lists file types supported for use in Sndstream.

Table 2 Supported File Types

File Type	Description
WAV (mono and stereo)	An uncompressed PCM format.
ATRAC9™ (mono and stereo)	A Sony proprietary compressed file format.
VAG/HE-VAG (mono and stereo)	A Sony proprietary ADPCM-based file format.
XVAG	An extensible file format; used for multi-Layer Streams.

3 Configuration, Initialization, and Shutdown

This chapter explains how to configure, initialize, and shutdown Sndstream.

Setting SceScreamSndStreamPlatformInit Structure Members

The `SceScreamSndStreamPlatformInit` data structure stores the platform-specific arguments required for initializing the Sndstream runtime component. You initialize the structure with default values by calling `sceScreamFillDefaultPlatformInitArgs()`. You may need to reset certain members according to your own system or application requirements. A complete description of `SceScreamSndStreamPlatformInit` structure members can be found in the *Sndstream Library Reference*. The following members merit some additional explanation:

- **size** – The *`size`* member must be set to the size of the structure in bytes. Failure to correctly set this member when initializing `SceScreamSndStreamPlatformInit` results in an error being returned by `sceScreamFillDefaultPlatformInitArgs()`. An incorrect setting can sometimes result from mismatched libraries referenced in the header. For example, if a library is updated after the application code is written, the size of the structure may have changed. The code example under Initializing Sndstream below shows the correct way to set the *`size`* member.
- **streaming_thread_priority** and **parsing_thread_priority** – Specify thread priority as a 32-bit integer. On the Windows platform, however, 32-bit integer values must be interpolated to fall within a narrower permissible range. As a side-effect of this interpolation, in certain cases, a small increase in the specified value has no impact on Windows (due to the interpolation), but can still make a noticeable difference on the PlayStation®4 and PlayStation®Vita platforms.
- **flags** – Sndstream offers initialization options that provide alternatives to default system behaviors. Currently, the supported Initialization Flags are:
  - `SCE_SCREAM_SND_SS_INIT_NO_PATH_COPY`: Initializes Sndstream to not allocate memory for file path storage.
  - `SCE_SCREAM_SND_SS_INIT_NO_MULTISTREAMS`: Initializes Sndstream without the capability to play multi-Layer Stream files.
- **extraStreamsForStealing** – Stream stealing is based on priorities specified in Bank contents or in the `SceScreamSndStartParams` structure's priority member when starting a Stream. Stream stealing involves starting a new Stream while a stolen Stream finishes, and therefore requires buffer management as well as voice management. For this reason, an additional internal structure is needed for every Bitstream being stolen. The memory cost per structure is only a few hundred bytes. This member specifies the number of additional internal structures to allocate for Stream stealing.

Setting Stream Handle Count

You set the maximum number of simultaneously playable Bitstreams in the `sceScreamInitStreaming()` function's *`handleCount`* parameter. Be sure to include any cross-fading Bitstreams in the overall count of simultaneously playing Bitstreams. For example, if your game has three simultaneous Bitstreams, but one of these Bitstreams sometimes cross-fades between two Stream files, then at the cross-fade points the game effectively has four simultaneous Bitstreams. And therefore you would set the `sceScreamInitStreaming()` function's *`handleCount`* parameter to 4.

If you are deploying multi-Layer, multi-Bitstream Stream files in your game, be sure to take account of the number of Bitstreams contained in your XVAG files. For further details, see Constraints on the Number of Bitstreams in the "Working with Multi-Layer Streams" chapter.

Setting Buffer Size

You set the Sndstream buffer size in the `sceScreamInitStreaming()` function's *`bufferSize`* parameter. The specified size applies to all Sndstream buffers, and therefore must be chosen with a worst-case scenario in mind. Sndstream buffer size has several dependencies, including sampling rate, compressed data format, and so on. Table 3 outlines these dependencies.

Table 3 Dependencies

Dependency	Description
Sampling rate	Higher sampling rates require greater buffer sizes as a result of having to stream more data per second. For example, a Stream playing an audio file with a 48 KHz sampling rate requires a larger buffer size than another Stream playing a file with a 44.1 KHz sampling rate.
Compressed data format	In Sndstream, ATRAC9™ compressed data is passed on to the underlying synthesizer for decoding. Therefore the compression ratio can factor into the required buffer size. For example, data in a 10:1 compression ratio would require a buffer one tenth the size of data in an uncompressed PCM format.
Media throughput	The throughput of the medium from which the audio file is being streamed also affects buffer size. Seek time is determined not only by the time required for the read head to locate the requested data, but also by other similarly-timed data reads (for example, of graphics, level data files, and so on).

A Rule-of-Thumb Formula

Given a worst-case seek time, you can estimate optimal buffer size using the following rule-of-thumb formula:

$$seekTime \times dataRate$$

where *seekTime* represents a worse-case scenario seek time (in seconds); and *dataRate* represents the MP3 data rate of the file to be streamed (in bytes per second).

Example

Suppose you are streaming 128-Kb/s ATRAC9™ files and your worst-case seek time is 2 seconds. First, you must convert your ATRAC9™ data rate from Kilo-bits-per-second (Kb/s) to Kilo-Bytes-per-second (KB/s). The calculations are as follows.

128 Kb/s = 16 KB/s

2.0 × 16 KB = 32 KB (that is, a buffer size of 32 KB)

Note: The formula serves as a good starting point. In practice however, you need to tune buffer sizes by auditioning your game and monitoring for possible artifacts in its audio streams.

Maximum Number of Simultaneous Streams

In Sndstream, the maximum number of Streams you can simultaneously play is simply the number of handles you specify when initializing Sndstream using the `sceScreamInitStreaming()` function, as discussed in Setting Stream Handle Count.

Initializing Sndstream

You must initialize Sndstream before use. Before initializing Sndstream, however, you must first initialize Scream. For information on initializing Scream, see the "Configuration, Initialization, and Shutdown" chapter in the *Scream Library Overview*.

You initialize Sndstream using the `sceScreamInitStreaming()` function, which has three parameters:

- **handleCount** – The number of streaming handles to create. Set this parameter to the maximum number of simultaneously playing streaming files that will occur. For details, see Setting Stream Handle Count.
- **bufferSize** – The size, in bytes, of one streaming buffer. See Setting Buffer Size for further details.
- **args** – A pointer to an initialized SceScreamSndStreamPlatformInit data structure. See Setting SceScreamSndStreamPlatformInit Structure Members for further details.

To summarize, the steps for initializing Sndstream are as follows:

(1) Fill the `SceScreamSndStreamPlatformInit` structure with default values by calling the `sceScreamFillDefaultPlatformInitArgs()` function.

(2) Reset any `SceScreamSndStreamPlatformInit` members in accordance with your own system or application requirements; see Setting SceScreamSndStreamPlatformInit Structure Members above.

(3) Call `sceScreamInitStreaming()`.

The following code example shows a basic procedure to initialize Sndstream:

```
SceScreamSndStreamPlatformInit initargs;
memset(&initargs, 0, sizeof(SceScreamSndStreamPlatformInit));
initargs.size = sizeof(SceScreamSndStreamPlatformInit);

// Initialize start args with default values
sceScreamFillDefaultPlatformInitArgs(&initargs);

// Set Sndstream threads to utilize CPU 2 only (zero-based index)
initargs.thread_affinity = 1;

// Divide the stream buffer into 8 sub buffers.
initargs.subBufferCount = 8;

err = sceScreamInitStreaming(2,          // max. of 2 simultaneous streaming inputs
                  (128*1024),    // stream buffer size is 128 KB
                  &initargs);
```

Closing Sndstream

To close Sndstream, call the `sceScreamCloseStreaming()` function with no arguments. `sceScreamCloseStreaming()` de-allocates all memory that was allocated to Sndstream. If any Streams are currently active, `sceScreamCloseStreaming()` calls `sceScreamStopAllStreams()`, which blocks the application thread until all Streams are stopped.

4 Working with System Globals

This chapter explains global system functionality.

Using Custom File I/O Functions

By default, Sndstream uses FIOS for file I/O on the PlayStation®Vita and PlayStation®4 platforms. It is also possible to override the FIOS functions with custom file I/O functions.

Sndstream allows you to override specific file I/O operations using custom function prototypes. In Sndstream, all data reading is asynchronous. Sndstream streams audio data using asynchronous file I/O executed during the Scream tick, that is during the time Scream interrupts to perform its processing, on the order of every 240th of a second. Other file I/O functionality, such as pre-parsing audio file headers, executes synchronously. The operations that can be overridden and their corresponding function prototypes are shown in Table 4.

Table 4 File I/O Operations and their Corresponding Function Prototypes

Operation	Mode	Function Prototype
File open	Synchronous	`SceScreamSndStreamFileOpenFunction`
File information callback	Synchronous	`SceScreamSndStreamFileInfoCBFunction`
File seek	Synchronous	`SceScreamSndStreamFileSeekFunction`
File read	Synchronous	`SceScreamSndStreamFileReadFunction`
File close	Synchronous	`SceScreamSndStreamFileCloseFunction`
File open	Asynchronous	`SceScreamSndStreamFileAsyncOpenFunction`
Is file open complete?	Asynchronous	`SceScreamSndStreamFileAsyncIsOpenCompleteFunction`
Initialize Bitstream data	Synchronous	`SceScreamSndStreamFileAsyncOpenBitstreamFunction`
File read	Asynchronous	`SceScreamSndStreamFileAsyncReadFunction`
Is file read complete?	Asynchronous	`SceScreamSndStreamFileAsyncIsReadCompleteFunction`
Reset Bitstream data	Synchronous	`SceScreamSndStreamFileAsyncCloseBitstreamFunction`
File close	Synchronous	`SceScreamSndStreamFileAsyncCloseFunction` **Note:** Synchronously closes a file that was opened for asynchronous reading.

When creating custom alternatives to these operations, your functions must adhere to the input and output parameter data types as specified in the Function Prototypes section of the *Sndstream Library Reference*.

Having created custom file I/O functions, perform the remaining steps to ensure that these functions substitute the FIOS functions:

(1) Store the addresses of your custom file I/O functions as the corresponding members of the `SceScreamSndFileInterface` data structure.

(2) After initializing Sndstream, and before calling any other Sndstream functions, call `sceScreamSetDefaultFileInterface()` with the initialized `SceScreamSndFileInterface` data structure as its argument.

Note: If you specify custom file I/O functions, you must set all `SceScreamSndFileInterface` structure members, except for the optional `SceScreamSndStreamFileInfoCBFunction`.

Tuning Your File Read System

Sndstream calls an optional custom file-information callback function once per Stream file, immediately after the file is opened. If provided, your custom file-information callback function must adhere to the `SceScreamSndStreamFileInfoCBFunction` prototype. Its address must be set using the *`m_pFileInfoCB`* member of the `SceScreamSndFileInterface` structure. Finally, to register the custom file information callback, you call the `sceScreamSetDefaultFileInterface()` function; see Step 2 under Using Custom File I/O Functions. Sndstream passes file handle, data rate, and loop count information to your callback function, allowing you to tune your file read system.

5 Starting a Stream

In Sndstream there are two ways to start a Stream:

- by reference to a file path, using the `sceScreamStartStream()` function
- by reference to a (previously created) file token, using the `sceScreamStartStreamByFileToken()` function

Even if you choose to reference a Stream file by its path, Sndstream parses the file's header and creates a token automatically using a single background thread. For this reason, it may be more efficient to create file tokens explicitly, either at load time or build time. See Creating File Tokens in the "Working with File Tokens" chapter for details. The `sceScreamStartStream()` function and referencing a file by its path are still supported.

This chapter explains both methods of starting a Stream, along with the attendant API structures and functions.

Starting a Stream by Reference to a File Path

You can start a Stream by reference to a file path using the `sceScreamStartStream()` function. The function has three parameters:

- **fileParams** takes a pointer to a `SceScreamSndFileParams` structure, which includes an embedded `SceScreamSndFileInterface` structure.
- **startParams** takes a pointer to a `SceScreamSndStartParams` structure, which includes an embedded `SceScreamSndBitstreamParams` structure and an embedded Scream `SceScreamSoundParams` structure.
- An output destination for the Stream, which defaults to the master buss.

The SceScreamSndFileParams and SceScreamSndFileInterface Structures

The `SceScreamSndFileParams` structure allows you to specify Stream-specific file settings. The embedded `SceScreamSndFileInterface` structure allows you to specify Stream-specific custom file I/O settings. These structures and their members are described in the *Sndstream Library Reference*. For more information on setting `SceScreamSndFileInterface`, see Using Custom File I/O Functions in the "Working with System Globals" chapter.

The SceScreamSndStartParams and SceScreamSndBitstreamParams Structures

The `SceScreamSndStartParams` structure allows you to specify parameter values required for starting a Stream. The (embedded) `SceScreamSndBitstreamParams` structure allows you to specify Bitstream-specific *`gain`*, *`azimuth`*, and *`focus`* parameter values. These structures and their members are described in the *Sndstream Library Reference*.

Note: Streams containing multiple Bitstreams are not supported in this release. Set the `SceScreamSndBitstreamParams` *`mask`* member to NULL when using the `SceScreamSndBitstreamParams` structure as a value for the `SceScreamSndStartParams` *`bitstreamParams`* member.

Smart Pan

The `SceScreamSndStartParams` *`flags`* member allows you to specify optional behaviors with which you can initialize a Stream. See "Stream Initialization Constants" in the *Sndstream Library Reference* for details on these behaviors. One of these optional behaviors warrants additional explanation – smart pan.

Smart pan is a simple distance simulation mechanism based on gain levels; it provides an alternative to a mechanism based on three-dimensional coordinates. For smart pan to work, Stream gain changes must be controlled directly by the Scream runtime. With smart pan enabled, as gain decreases below 0.85, pan

values begin to narrow, converging on 0 as gain reaches 0.15 and below. Conversely, as gain increases above 0.15, pan values widen, expanding to a specified maximum setting as gain reaches 0.85 and above. Smart pan can be useful for sound effects that are attached to a localized area within a game environment, such as a waterfall.

For example, a waterfall sound might be implemented as a stereo Stream file, panned wide to produce an effect that surrounds the listener. When the listener is close to the waterfall (and gain is high), you want maximum stereo panning width to create the surround effect. But as the listener moves away from the waterfall (and gain is reduced), smart pan contracts channel panning widths to create an increasingly directional effect. As well as setting this option in the `SceScreamSndStartParams` *`flags`* member, you should also ensure that the (Scream) `SceScreamSoundParams` *`azimuth`* member is not set to a specific Output Speaker Target. For best results, pan individual Stream channels wide. These settings are heard as the maximum pan values when gain is ≥ 0.85. Control Stream gain through the Scream API's `SceScreamSoundParams` *`gain`* member.

The sceScreamStartStream() Function

The `sceScreamStartStream()` function returns the handle of an initialized Stream. This handle can be used as input to a range of other Sndstream and Scream API function calls, allowing you to manipulate the Stream and retrieve status information.

Starting a Stream by Reference to a File Token

You can start a Stream by reference to a file token using the `sceScreamStartStreamByFileToken()` function. The function has four parameters:

- A file token.
- A pointer to a `SceScreamSndStreamQueueParams` structure, specifying playback information.
- A pointer to a `SceScreamSndStartParams` structure, initialized with parameter values for the new Stream; including an embedded `SceScreamSndBitstreamParams` structure and an embedded Scream `SceScreamSoundParams` structure.
- An output destination for the Stream, which defaults to the master buss.

Obtaining a File Token

To create a file token you pre-parse an audio file using the `sceScreamParseStreamFile()` function. See the Working with File Tokens chapter for details.

The SceScreamSndStreamQueueParams Structure

The `SceScreamSndStreamQueueParams` structure allows you to specify playback information for starting or queuing a Stream by reference to a file token. The structure's members are described as follows:

- **loopCount** – Specifies a number of additional loops to play. That is, 0 to play once without looping, 1 to play twice, 2 to play 3 times, and so on. To loop indefinitely, use the `SCE_SCREAM_SND_SS_LOOP_INFINITE` constant; to loop until another file has been queued on the handle, use `SCE_SCREAM_SND_SS_LOOP_TILL_QUEUED`.
- **startSecond** – Offset number of seconds into the file at which point to start playback. Used in cases where the desired audio data starts other than at the beginning of the specified file. Expressed in seconds.

Note: Applies to WAV and VAG file formats only.

The SceScreamSndStartParams and SceScreamSndBitstreamParams Structures

Use of these structures is the same as for starting a Stream by reference to a file path. See The SceScreamSndStartParams and SceScreamSndBitstreamParams Structures section above for details.

The sceScreamStartStreamByFileToken() Function

The `sceScreamStartStreamByFileToken()` function returns the handle of an initialized Stream. This handle can be used as input to a range of other Sndstream and Scream API function calls, allowing you to manipulate the Stream and retrieve status information.

> **Note:** The `sceScreamStartStreamByFileToken()` function copies the information from the file token. So you can safely delete the file token if you do not plan to re-use it.

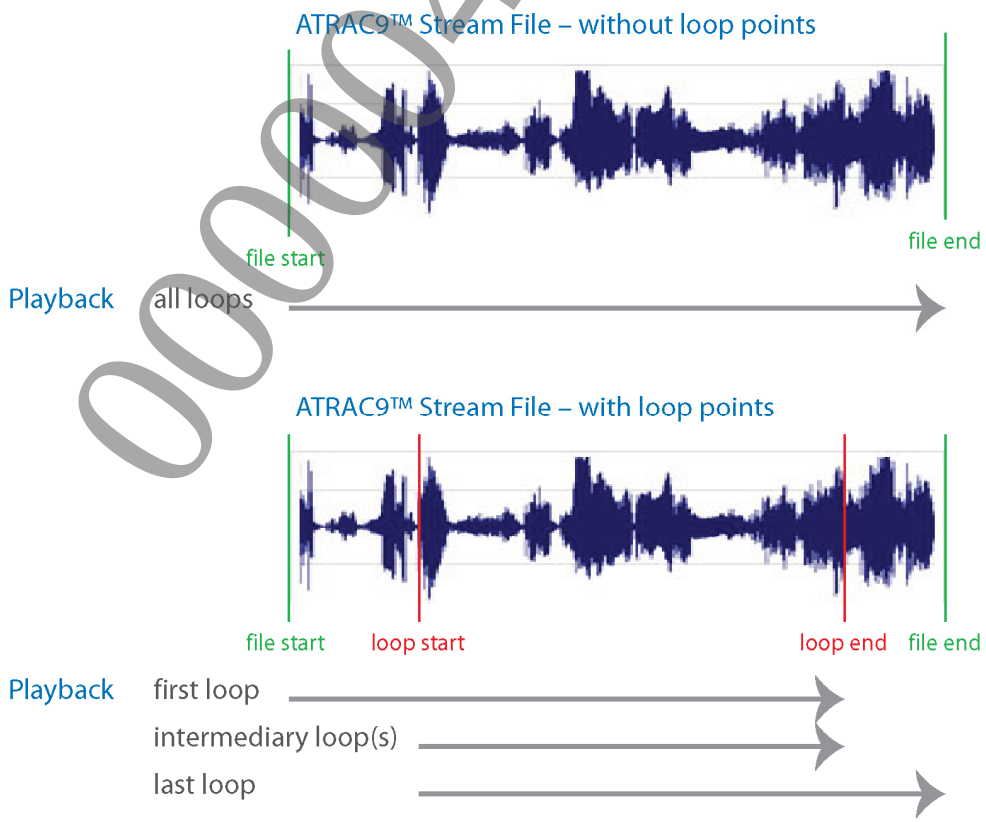
Working with Embedded Loop Points in Stream Files

Using `at9tool`, a command-line utility available with the PlayStation®Vita and PlayStation®4 SDKs, audio designers can specify loop start and end points within ATRAC9™ files. Sndstream recognizes embedded loop points in ATRAC9™ files, and, if found, modifies its looping playback behavior accordingly.

Setting Stream looping behavior using the Sndstream API varies according to whether you are starting a Stream by reference to a file path or by reference to a file token. When starting a Stream by reference to a file path, you set looping behavior in the `SceScreamSndFileParams` structure's *`loopCount`* member. When starting a Stream by reference to a file token, you set looping behavior in the `SceScreamSndStreamQueueParams` structure's *`loopCount`* member. In either case, the specification of looping behavior is the same. See The SceScreamSndStreamQueueParams Structure above for details.

If Sndstream encounters embedded loop points in a Stream file, instead of looping around the entire file from start to end, looping takes place between the loop points. Playback always starts from the beginning of a file, but if a loop end point is encountered before the end of the file, playback loops back from there instead of playing through to the end. Similarly, on looping back, if a loop start point is encountered after the start of the file, playback continues from there instead of from the start of the file. On the last loop, if there is data between the loop end point and the end of the file, it is played. Figure 3 shows the difference between looping playback with and without embedded loop points.

Figure 3 Stream File Looping Playback, With and Without Loop Points



6 Synchronizing Stream Transitions, Overlays, and Scream Sounds

This chapter explains how to start a new Stream as a synchronized transition or overlay with an existing Stream, and how to coordinate synchronized playback of Scream Sounds.

Starting a Stream as a Transition from or Overlay with an Existing Stream

Using the `sceScreamStartStreamFromTransition()` function, you can start a new Stream as a synchronized transition from an existing Stream or as an overlay in synchronization with an existing Stream.

Setting a Sync Clock Stream

The presence of a sync clock Stream is necessary for coordinating a Stream transition (or overlay) using the `sceScreamStartStreamFromTransition()` function. There is only one sync clock Stream at a time, and `sceScreamStartStreamFromTransition()` sets it. The sync clock Stream may or may not be the existing Stream from which you are transitioning. In either case, it must have an associated MIDI file specifying appropriate tempo and meter information for the Stream from which you are transitioning. You set a sync clock Stream by applying the `SCE_SCREAM_SND_SS_START_SYNC_CLOCK` flag to the `SceScreamSndStartParams` *`flags`* member; `sceScreamStartStreamFromTransition()` contains a `SceScreamSndStartParams` parameter.

> **Note:** If transitioning from a Stream that is set as the current sync clock Stream, the current sync clock Stream terminates at the transition point. Transition points in the Stream may be set in a variety of ways. They may arise from Bank contents set by designers. You can also set transition points in Sndstream, as described in Setting SceScreamSndSyncParams Structure Members below. You have three options for how to manage this scenario. See the Notes section for the `sceScreamStartStreamFromTransition()` function in the *Sndstream Library Reference* for details.

Retrieving the Current Sync Clock Stream's Handle

You can retrieve the handle of the current sync clock Stream using the `sceScreamGetCurrentSyncClockStreamHandle()` function. The function has no parameters, and simply returns the sync clock Stream handle if there is one. If there is no handle, it returns 0.

MIDI File Specifications

A MIDI file used for the sync clock Stream must share the same path and base file name as the Stream's associated audio file, but with a `.mid` extension replacing – not appended to – the audio file extension. For example, an audio file named `/usr/sounds/musicClip1.vag` would be paired with a MIDI file named `/usr/sounds/musicClip1.mid`.

Further details on the contents of a MIDI file used to coordinate Stream transition can be found in the "Stream Transitions" chapter of the *Scream Tool Help*.

The sceScreamStartStreamFromTransition() Function and its Parameters

You start a new Stream as a coordinated transition from an existing Stream using the `sceScreamStartStreamFromTransition()` function. This mechanism is designed for music-oriented Stream transitions. It coordinates the transition on a musical boundary, such as measures and measure subdivisions. This feature provides a degree of game-dependent interactivity to a sequential playback of a range of music clips.

The `sceScreamStartStreamFromTransition()` function has five parameters. The first parameter, *`transitionHandle`*, is the handle an existing Stream that you want to transition from or play along with. The remaining four parameters are pointers to the following structures:

- `SceScreamSndSyncParams` – Stores synchronization properties for Stream transitions or synchronized playback.
- `SceScreamSndTransitionParams` – Stores properties for a Stream transition.
- `SceScreamSndFileParams` – Stores Stream file parameter values.
- `SceScreamSndStartParams` – Stores the parameter values required for starting a Stream.

Considerations regarding the `SceScreamSndFileParams` and `SceScreamSndStartParams` structures were discussed in the Starting a Stream chapter and are identical in regard to starting a Stream as a transition from an existing Stream. The `SceScreamSndSyncParams` and `SceScreamSndTransitionParams` structures are discussed below.

Setting SceScreamSndSyncParams Structure Members

The `SceScreamSndSyncParams` structure has three members: *`syncFlags`*, *`syncUnit`*, and *`unitMultiple`*.

syncFlags

In the *`syncFlags`* member you specify transition behaviors – one, both, or neither of the following flags:

- `SCE_SCREAM_SND_SYNC_FLAG_START_IF_NO_CLOCK` – Allows the new Stream to still play even if there is no sync clock Stream to which to synchronize.
- `SCE_SCREAM_SND_SYNC_FLAG_START_IF_CLOCK_ENDS` – Allows the new Stream to still play even if the sync clock Stream terminates before a legal sync point is reached.

Set the *`syncFlags`* member to 0 if, in the absence of a sync clock Stream, you do not want a synchronized Stream to play (that is, out of synchronization).

syncUnit and unitMultiple

The *`syncUnit`* and *`unitMultiple`* members allow you to specify synchronization points for a Stream transition or overlay.

> **Note:** You can define synchronization points either in the content (that is, the sync clock Stream's associated MIDI file) or in the API (using the *`syncUnit`* and *`unitMultiple`* members). To specify synchronization points as defined in the content, set the *`syncUnit`* member to `SCE_SCREAM_SND_SYNC_UNIT_CONTENT`. In this case, values for *`syncUnit`* and *`unitMultiple`* are taken from transition markers contained in the sync clock Stream's associated MIDI file. Any value set for the *`unitMultiple`* member is ignored.

If specifying Stream transition synchronization points from the API, you set the *`syncUnit`* member to a value other than `SCE_SCREAM_SND_SYNC_UNIT_CONTENT`. You then define synchronization points as an integer number of multiples (*`unitMultiple`*) of a basic synchronization unit (*`syncUnit`*). The remaining options for the *`syncUnit`* member effectively override any synchronization points that may be specified in MIDI markers contained in the sync clock Stream's MIDI file. These options are described as follows:

- `SCE_SCREAM_SND_SYNC_UNIT_CLOCK` – The synchronization point falls on a sync clock boundary; of which there are 24 per quarter-note (expressed in the constant `SCE_SCREAM_SND_SYNC_CLOCKS_PER_QUARTER`).
- `SCE_SCREAM_SND_SYNC_UNIT_BEAT` – The synchronization point falls on a beat boundary.
- `SCE_SCREAM_SND_SYNC_UNIT_MEASURE` – The synchronization point falls on a measure boundary.
- `SCE_SCREAM_SND_SYNC_UNIT_MARKER` – The synchronization point falls on a MIDI marker boundary (contained in the sync clock Stream's associated MIDI file).

You specify an integer value for the *unitMultiple* member. With appropriate settings for both members, synchronization points occur at every *unitMultiple* of the *syncUnit* value. For example, with *syncUnit* set to SCE_SCREAM_SND_SYNC_UNIT_BEAT, and *unitMultiple* set to 2, synchronization points occur at every two beats.

For *syncUnit* values smaller than a quarter-note, the resolution is in sync clocks, which are 1/24th subdivisions of a quarter-note, defined as SCE_SCREAM_SND_SYNC_CLOCKS_PER_QUARTER. For example, an eighth-note is 12 sync clocks; a triplet-eighth-note is 8 sync clocks; and a sixteenth-note is 6 sync clocks, and so on. The sync clock macros may be of help in calculating multiples of quarter-notes and quarter-note subdivisions in terms of sync clocks. See "Synchronization Constants" in the *Sndstream Library Reference* for details on the following macros:

- SCE_SCREAM_SND_UNIT_CLOCK_MULTIPLE_QUARTER_NOTE
- SCE_SCREAM_SND_UNIT_CLOCK_MULTIPLE_EIGHTH_NOTE
- SCE_SCREAM_SND_UNIT_CLOCK_MULTIPLE_SIXTEENTH_NOTE

Setting SceScreamSndTransitionParams Structure Members

The SceScreamSndTransitionParams structure has five members: *transitionMode*, *fadeInTime*, *fadeInGain*, *fadeOutTime*, and *fadeOutGain*.

transitionMode

The sceScreamStartStreamFromTransition() function allows you to either start a new Stream as a synchronized transition from an existing Stream or to overlay a new Stream in synchronization with an existing Stream. The choice of whether to transition from or overlay with an existing Stream is set in the *transitionMode* member. The Transition Mode Constants actually provide three options:

- SCE_SCREAM_SND_TRANSITION_MODE_FADEOUT_MASTER – The existing master Stream fades out in accordance with the *fadeOutTime* and *fadeOutGain* members.
- SCE_SCREAM_SND_TRANSITION_MODE_KEYOFF_MASTER – The existing master Stream keys-off (that is, enters the Release stage of an ADSR setting, rather than fading out) at the transition point.
- SCE_SCREAM_SND_TRANSITION_MODE_PLAY_WITH_MASTER – A new transitioned Stream plays along with a master (existing) Stream as the latter continues.

fadeInTime and fadeInGain

The *fadeInTime* and *fadeInGain* members specify the fade-in time of the transitioned Stream, expressed in seconds, and target gain level of the transitioned Stream on completion of its fade-in.

fadeOutTime and fadeOutGain

The *fadeOutTime* and *fadeOutGain* members are only applicable if *transitionMode* is set to SCE_SCREAM_SND_TRANSITION_MODE_FADEOUT_MASTER; otherwise, they are ignored. These members specify the fade-out time of the existing Stream, expressed in seconds, and target gain level of the existing Stream on completion of its fade-out.

Transitioned Streams and Premature Unpausing

When initializing a new Stream as a transition from an existing Stream, the new Stream is initialized immediately, but paused until it is buffered *and* until a permissible synchronization point is reached in the existing Stream. At the transition point, the new Stream is unpaused. However, if you call the Scream function sceScreamContinueSound() with the handle of the new Stream before it has started, it starts as soon as it is buffered, that is, without waiting for the synchronization point.

Synchronizing Playback of Scream Sounds

In conjunction with a sync clock Stream you can coordinate synchronized playback of Scream Sounds.

Two Sndstream functions allow you to play a Scream Sound in synchronization with the sync clock Stream:

- `sceScreamPlaySoundSyncedByIndexEx()` – Plays a Scream Sound, by reference to its index, in synchronization with the sync clock.
- `sceScreamPlaySoundSyncedByNameEx()` – Plays a Scream Sound, by reference to its name, in synchronization with the sync clock.

You can use the `sceScreamGetCurrentSyncClockStreamHandle()` function to determine if there is a currently active sync clock Stream. If there is none, you must start a Stream as the sync clock Stream. See Setting a Sync Clock Stream for details.

The `sceScreamPlaySoundSyncedByIndexEx()` and `sceScreamPlaySoundSyncedByNameEx()` functions both have four parameters. The *`bank`*, *`params`*, and *`syncParams`* parameters are common to both functions, whereas the second parameter is function-specific, specifying the Sound to play by reference to its index or name.

The *`bank`* parameter is a Scream *`SceScreamSFXBlock2`* pointer to the Bank that contains the Sound to play. See the *Scream Library Reference* documents.

The *`params`* parameter is a pointer to a Scream `SceScreamSoundParams` structure initialized with Sound parameter settings. See the *Scream Library Reference* documents for details about this structure.

The *`syncParams`* parameter is a pointer to an initialized `SceScreamSndSyncParams` structure containing synchronization properties. See Setting SceScreamSndSyncParams Structure Members for details.

7 Manipulating an Active Stream

Inserting a File into the Queue of an Existing Stream

As well as initiating playback of an audio file as a new Stream (with `sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, or `sceScreamStartStreamFromTransition()`), you can also insert an audio file into the queue of an existing Stream. You do this using the `sceScreamQueueToStream()` function, which allows you to insert a file into any position of an active Stream's queue. This can be useful, for example, for dynamic stitching of game-dependent dialog content.

The `sceScreamQueueToStream()` function takes four arguments: *`queueHandle`*, *`queueIndex`*, *`fileParams`*, and *`startParams`*. The *`fileParams`* and *`startParams`* parameters take pointers to the respective `SceScreamSndFileParams` and `SceScreamSndStartParams` structures. Their usage is identical here as in the `sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, and `sceScreamStartStreamFromTransition()` functions; see The SceScreamSndFileParams and SceScreamSndFileInterface Structures and The SceScreamSndStartParams and SceScreamSndBitstreamParams Structures sections in the "Starting a Stream" chapter for details.

In the *`queueHandle`* parameter you specify the handle of an active Stream into the queue of which to insert the new audio file. The Stream handle is a value returned by the `sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, or `sceScreamStartStreamFromTransition()` functions.

In the *`queueIndex`* parameter you specify a zero-based index indicating a position in the queue at which point to insert the new file. A Stream can have up to seven files attached to its queue; expressed in the constant `SCE_SCREAM_SND_FILE_QUEUE_MAX`. The range of permissible index values is from `SCE_SCREAM_SND_QUEUE_INDEX_HEAD` (a constant expressing the position at the front of the queue, equivalent to 0) to `SCE_SCREAM_SND_QUEUE_INDEX_TAIL` (a constant that expresses the position at the end of the queue, regardless of its current length).

For an example of dynamically stitching a file onto the queue of an active Stream, see the Sndstream Stitch sample program in the "Introduction" chapter.

Adjusting Playback of a Stream's Currently Playing File

You can dynamically update a Stream's remaining loop count. You can also move the playback position within the file being played.

Updating Remaining Loop Count

You can update the remaining loop count of a Stream's currently playing file using the `sceScreamSetStreamFileLoopingCount()` function. The function takes two arguments: *`handle`* and *`loopCount`*. You specify the handle of the Stream for which you want to set a new loop count in the *`handle`* parameter. This is a value returned from the `sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, or `sceScreamStartStreamFromTransition()` functions. In the *`loopCount`* parameter you can either set a numeric value or use one of the following constants:

- `SCE_SCREAM_SND_SS_LOOP_INFINITE` – Causes the currently playing file to loop indefinitely.
- `SCE_SCREAM_SND_SS_LOOP_TILL_QUEUED` – Causes the currently playing file to loop until a new file is added to the empty queue on the same handle.

If setting a numeric value, the value indicates the number of complete loops to play following the current cycle. For example, 1 specifies playback of one more complete loop after the current cycle; 2 specifies playback of two more complete loops after the current cycle, and so on.

Note: While a Stream can have multiple queued files, the new loop count is applied only to the currently playing file.

See also: Working with Embedded Loop Points in Stream Files in the "Starting a Stream" chapter.

Usage Scenario

Suppose a looping music Stream is in its last repetition when game contexts demand a further continuation of this playing music track. You can add further repetitions or temporarily set the file to loop indefinitely. Conversely, if a file is already set to loop indefinitely and game contexts demand a change of musical feel, you have two options depending on the status of the Stream's queue:

- If the Stream already has at least one file in its queue, you can set *loopCount* to 0. This causes the Stream to move onto the next queued file at the end of its current cycle. You can use the `sceScreamGetStreamFileLoopingCount()` function to get the queue count.
- If the Stream has an empty queue, you can set *loopCount* to `SCE_SCREAM_SND_SS_LOOP_TILL_QUEUED`, and then add a new file to the Stream's queue. This causes the Stream to move onto the newly queued file at the end of its current cycle. See Inserting a File into the Queue of an Existing Stream above for details.

Moving Playback Position

You can move the playback position within a currently playing file using the `sceScreamCueStreamToTime()` function. This can be useful if the audio file has multiple sections and you want to immediately move to a different section and continue playback (perhaps due to changing game contexts).

The `sceScreamCueStreamToTime()` function takes two arguments: *handle*, and *seconds*. You specify the handle of the Stream for which you want to move the playback position in the *handle* parameter; this is a value returned from the `sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, or `sceScreamStartStreamFromTransition()` functions. In the *seconds* parameter you set a floating point time offset (in seconds) from the beginning of the file that specifies the new playback position.

Consultation Point: Care should be taken when using the `sceScreamCueStreamToTime()` function to avoid jumping to a position that might produce audible discontinuities. You should consult with your audio designer to determine an appropriate time offset value to which to move the playback position.

Note: Time offsets for VAG files are rounded to the nearest 24-sample boundary.

Setting Layer Parameters Collectively

You can set and retrieve Layer parameter values collectively. You can also set automated incremental updates to Layer parameter values collectively. Whether working with a standard (single-Layer/Bitstream) Stream or a multi-Layer/Bitstream Stream, the procedure is essentially the same. The only difference is the depth of the arrays used in the `SceScreamSndBitstreamParams` structure members that store Layer parameter values.

Note: To set Layer parameters individually, you must first obtain individual Layer handles. You can then use these handles with the Scream `sceScreamSetSoundParamsEx()` function to set any parameter, including synthesizer parameters, on an individual Layer. You can also use the Scream `sceScreamAutoGain()` and `sceScreamAutoPan()` functions to automate incremental updates to gain or pan parameters on individual Layers. For further details, see Retrieving Individual Layer Handles in the "Working with Multi-Layer Streams" chapter.

Instantaneous Parameter Setting and Retrieval

You can update Layer parameters collectively at the same time using the `sceScreamSetStreamLayerParams()` function.

For the *`handle`* parameter of this function, you specify the handle of a Stream that contains the Layers you want to set. This value is returned from the Stream-starting functions `sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, and `sceScreamStartStreamFromTransition()`.

For the *`params`* parameter you specify a pointer to a `SceScreamSndBitstreamParams` structure, specifying gain, azimuth, and focus values for every Layer in a Stream. The `SceScreamSndBitstreamParams` structure's *`gain`*, *`azimuth`*, and *`focus`* members store parameter values in arrays, the depth of which being equal to the number of Layers contained in a target Stream. Use of the `SceScreamSndBitstreamParams` structure in this context is identical to its use when starting a Stream. For further details, see The SceScreamSndStartParams and SceScreamSndBitstreamParams Structures in the "Starting a Stream" chapter.

For the *`layerCount`* parameter you specify the number of Layers being set. This value indicates the *`length`* of the arrays used in the `SceScreamSndBitstreamParams` structure pointed to in the `params` argument.

You can retrieve Layer parameter values collectively, using the `sceScreamGetStreamLayerParams()` function. This function retrieves current gain, azimuth, and focus parameter values from all Layers associated with a Stream handle. The function stores retrieved parameter values in a `SceScreamSndBitstreamParams` structure.

Automated Incremental Settings

You can automate incremental changes to Layer gain and azimuth parameters collectively using the `sceScreamAutoStreamLayerParams()` function. This function is very similar to the Scream automated parameter change functions: `sceScreamAutoPan()`, `sceScreamAutoPitchTranspose()`, `sceScreamAutoPitchBend()`, and `sceScreamAutoGain()`. Its parameters are described as follows:

- **handle** — The handle of a Stream upon which to apply automated parameter changes.
- **params** — A `SceScreamSndBitstreamParams` structure, containing an array of gain and azimuth parameter values, one for each Layer.

> **Note:** In addition to gain and azimuth values, the `SceScreamSndBitstreamParams` structure also stores an array of focus parameter values. Focus values, however, can be set instantaneously only. That is, focus values cannot be incrementally automated, as with gain and azimuth values.

- **layerCount** — The number of Layers being automated. This value determines the length of the arrays used in the `SceScreamSndBitstreamParams` structure pointed to in the *`params`* argument. For further details, see the Working with Multi-Layer Streams chapter.
- **timeToTarget** — Time in which to gradually change the parameters to reach the target values specified in the `params` argument. Expressed in seconds, as an array of time values, one for each Layer.
- **behaviorFlags** — Optional parameter change behaviors. One or more of the Automated Parameter Change Flags (see the *Sndstream Library Reference* for details). Expressed as an array of constants, one or more for each Layer. For a more detailed discussion of the behavior flags (in particular, the *Revert If Active* flag), see "Applying Automated Changes to Parameter Values" section of the "Working with Sounds" chapter of the *Scream Library Overview*.

Stopping a Stream

To stop an individual Stream you use the Scream function `sceScreamStopSound()` using the Stream handles returned by the `sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, and `sceScreamStartStreamFromTransition()` functions as its argument. See the *Scream Library Reference* documents for further details.

Stopping All Streams

To stop all active Streams you use the `sceScreamStopAllStreams()` function. Note that this function blocks the calling thread until all Streams are stopped.

8 Retrieving Stream Information

You can retrieve a variety of information about a Stream, including file duration, time elapsed, time remaining, loop count, buffered status, channel count, voice level, and sample rate.

The Sndstream Play Info sample program serves as a useful source code reference for Stream information retrieval.

Retrieving Information from a Currently Playing Stream File

You can retrieve the duration, time elapsed, time remaining, and current voice level on a currently playing Stream file.

Duration

The `sceScreamGetStreamFileLengthInSeconds()` function retrieves the total duration of the currently playing file on a Stream. The function takes three arguments: *`handle`*, *`outSeconds`*, and *`outContext`*. You specify the handle of the Stream you want to retrieve a duration for in the *`handle`* parameter. This is a value returned from the Stream-starting functions (`sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, and `sceScreamStartStreamFromTransition()`). The remaining two parameters are pointers to output variables, used to store the retrieved information. The *`outSeconds`* variable receives the total duration. The *`outContext`* variable receives the user context value assigned to the file (by the application) in the *`userContext`* member of the `SceScreamSndFileParams` structure. You can set this parameter to `NULL` if user context information is not required.

Playback Position

The `sceScreamGetStreamFileLocationInSeconds()` function retrieves the current playback position of the currently playing file on a Stream. The function takes three arguments: *`handle`*, *`outLocation`*, and *`outContext`*. You specify the handle of the Stream you want to retrieve a playback position from in the *`handle`* parameter. This is a value returned from the Stream-starting functions (`sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, and `sceScreamStartStreamFromTransition()`). The remaining two parameters are pointers to output variables, used to store the retrieved information. The *`outLocation`* variable receives the current playback position. The *`outContext`* variable receives the user context value assigned to the file (by the application) in the *`userContext`* member of the `SceScreamSndFileParams` structure. You can set this parameter to `NULL` if user context information is not required.

Time Remaining

The `sceScreamGetStreamFileSecondsRemaining()` function retrieves the remaining duration of the currently playing file on a Stream. The function takes three arguments: *`handle`*, *`outSeconds`*, and *`outContext`*. You specify the handle of the Stream you want to retrieve a remaining duration for in the *`handle`* parameter. This is a value returned from the Stream-starting functions (`sceScreamStartStream()`, `sceScreamStartStreamByFileToken()`, and `sceScreamStartStreamFromTransition()`). The remaining two parameters are pointers to output variables, used to store the retrieved information. The *`outSeconds`* variable receives the remaining duration. The *`outContext`* variable receives the user-context value assigned to the file (by the application) in the *`userContext`* member of the `SceScreamSndFileParams` structure. You can set this parameter to `NULL` if user context information is not required.

Loop Count

The `sceScreamGetStreamFileLoopingCount()` function retrieves the number of loops currently assigned, and the number of completed loops of the currently playing file on a Stream. The function takes

four arguments: *handle*, *outSetting*, *outCount*, and *outContext*. You specify the handle of the Stream you want to retrieve a loop count for in the *handle* parameter. This is a value returned from the Stream-starting functions (sceScreamStartStream(), sceScreamStartStreamByFileToken(), and sceScreamStartStreamFromTransition()). The remaining three parameters are pointers to output variables, used to store the retrieved information.

The *outSetting* variable receives the loop setting currently assigned to the file (by the application) in the *loopCount* member of the SceScreamSndFileParams structure. Return values are as follows:

- SCE_SCREAM_SND_SS_LOOP_TILL_QUEUED – The file is set to loop until a new file is queued on the same handle.
- SCE_SCREAM_SND_SS_LOOP_INFINITE – The file is set to loop indefinitely.
- ≥0 – The file is set to loop a finite number of times. 0 means it was set to play once without looping; 1 means to play twice; 2 means to play 3 times, and so on.

The *outCount* variable receives the completed loop count of the playing file. This value starts at zero and is incremented each time the file reaches the end and starts over. If the value received by *outSetting* is greater than zero, you can determine the remaining number of complete loops still to play by subtracting *outCount* from *outSetting*.

The *outContext* variable receives the user context value assigned to the file (by the application) in the *userContext* member of the *SceScreamSndFileParams* structure. You can set this parameter to NULL if user context information is not required.

Level

The sceScreamGetStreamLevel() function retrieves the current voice level of a Stream. The function takes three arguments: *handle*, *rms* , and *linear*. You specify the handle of the Stream you want to retrieve a current voice level for in the *handle* parameter. This is a value returned by the Stream-starting functions (sceScreamStartStream(), sceScreamStartStreamByFileToken(), and sceScreamStartStreamFromTransition()). The *rms* parameter expresses the format in which voice level data is returned. Set *rms* to TRUE if you want an averaged RMS level, otherwise the function returns a value expressed as an instantaneous peak level. You specify desired units in which level values are expressed in the *linear* parameter. Set *linear* to TRUE if you want the result on a linear scale, otherwise the result is expressed in decibels (dB). Note also that to retrieve voice level data from a Stream, the Stream must have been initialized with the SCE_SCREAM_SND_SS_START_GET_VOICE_LEVEL option included in the SceScreamSndStartParams *flags* member.

Retrieving Stream Information and Queue Count

You can retrieve Stream information from a Stream, including buffered status, channel count, and sample rate. You can also retrieve the number of files queued on a Stream.

Stream Information

The sceScreamGetStreamInfo() function retrieves buffered status, Bitstream count, Bitstream channel count, and sample rate information from a Stream. The function takes five arguments: *handle*, *outBufferedStatus*, *outBitstreamCount*, *outChannelCount*, and *outSampleRate*. You specify the handle of the Stream from which to retrieve information from in the *handle* argument. This is a value returned from the Stream-starting functions (sceScreamStartStream(), sceScreamStartStreamByFileToken(), and sceScreamStartStreamFromTransition()). The remaining parameters are pointers to output variables used to store the retrieved information.

The *outBufferedStatus* variable receives the buffered status. If the Stream is buffered, a non-zero value is returned.

The *outBitstreamCount* variable receives the number of Bitstreams in the Stream. If the Stream is not buffered, a zero value is returned. You can set *outBitstreamCount* to NULL if Bitstream count

information is not required. Multiple Bitstreams are not supported in this release, so the returned value (if not zero) is always one.

The *outChannelCount* variable receives the number of channels per Bitstream. If the Stream is not buffered, a zero value is returned. You can set *outChannelCount* to NULL if channel count information is not required.

The *outSampleRate* variable receives the sample rate of the Stream. If the Stream is not buffered, a zero value is returned. You can set *outSampleRate* to NULL if sample rate information is not required.

Queue Count

The sceScreamGetStreamQueueCount() function retrieves the number of files queued on a Stream. The function takes two arguments: *handle* and *outQueueCount*. You specify the handle of the Stream you want to retrieve a queue count for in the *handle* parameter. This is a value returned from the Stream-starting functions (sceScreamStartStream(), sceScreamStartStreamByFileToken(), and sceScreamStartStreamFromTransition()). The *outQueueCount* parameter is a pointer to an output variable used to store the retrieved queue count. A value of zero can indicate either that the handle has no queued items (but may still be playing an active Stream), or is not active at all. Note that the maximum number of files that can be queued on a Stream handle is seven (in addition to the currently playing file); expressed in the constant SCE_SCREAM_SND_FILE_QUEUE_MAX.

9 Working with File Tokens

File tokens offer an efficient method by which you can reference Stream files. You can explicitly create file tokens at load time or at build time using the `sceScreamParseStreamFile()` function. If you choose not to explicitly create file tokens, Sndstream creates them automatically when you call the `sceScreamStartStream()`, `sceScreamStartStreamFromTransition()`, and `sceScreamQueueToStream()` functions at run time. Once created, file tokens can be used multiple times to start or queue Streams, including simultaneously playing Streams.

The Preparse example program serves as a useful source code reference for file token manipulation.

Creating File Tokens

You create file tokens at load time or build time by pre-parsing audio files intended for Streaming using the `sceScreamParseStreamFile()` function.

Creating File Tokens at Load Time

You create a file token at load time by calling the `sceScreamParseStreamFile()` function. Note that the function blocks until the specified audio file's header has been read. For this reason it may be impractical to create a large quantity of file tokens at load time, in which case, Creating and Storing File Tokens at Build Time may be preferable. The token returned by the `sceScreamParseStreamFile()` function represents the specified audio file, and contains information from the file's header that is required during playback. Technically, you can create a file token at any time before playing the file as a new Stream (with the `sceScreamStartStreamByFileToken()` function) or queuing it to an existing Stream (with the `sceScreamQueueToStreamByFileToken()` function).

The `sceScreamParseStreamFile()` function has two parameters:

- **parseParams –** a pointer to a `SceScreamSndStreamParseParams` structure, appropriately initialized for the associated Stream file.
- **errorCodePtr –** a pointer to a variable in which to receive an error code, or zero if no error occurs.

The `SceScreamSndStreamParseParams` structure includes many members in common with the `SceScreamSndFileParams` structure, the latter is used when starting a Stream with the `sceScreamStartStream()` function.

Note: A file token is an opaque pointer to an internal structure that contains information from the file's header. The memory required to store the information can be pre-allocated when Sndstream is initialized. The function `sceScreamInitStreaming()` allocates memory based on the value of the *`parsedFileCount`* member of the `SceScreamSndStreamPlatformInit` structure. If you set `SCE_SCREAM_SND_SS_FILE_ALLOCATION_OK` in the *`flags`* member of the `SceScreamSndStreamPlatformInit` structure, the required memory can be dynamically allocated if there is insufficient pre-allocated memory.

Creating and Storing File Tokens at Build Time

Because the `sceScreamParseStreamFile()` function blocks until the specified audio file's header has been read, creating file tokens at build time provides an efficient alternative to creating them at load time. When creating file tokens at build time you save them to disk using a file token storage. File tokens contained in a file token storage can be retrieved at run time using the `sceScreamGetFileTokenFromStorage()` function. If you choose to create file tokens at build time you may want to batch process the entire set of audio files intended for streaming during a game.

To create and store file tokens at build time:

(1) On the Windows platform, pre-parse a set of audio files using the `sceScreamParseStreamFile()` function. Save the resultant tokens into an array, keeping track of the indices and corresponding audio file paths.

(2) Create a file token storage using the `sceScreamCreateFileTokenStorage()` function, referencing the array of tokens created in Step 1.

(3) If the target platform has a different endianness from that of the build machine (as is the case when creating a file token storage on Windows for run time deployment on PlayStation®3), you can optionally byte reverse the storage using the `sceScreamByteReverseFileTokenStorage()` function.

(4) Save the file token storage to disk using any appropriate method.

(5) Optionally, free the memory associated with the file token storage using the `sceScreamDeleteFileTokenStorage()` function.

Retrieving File Tokens from a File Token Storage

The tasks of creating a set of file tokens representing audio files intended for streaming in your game, and creating a file token storage and populating it with file tokens should be done at build time.

For details, see Creating and Storing File Tokens at Build Time. At run time you can validate a file token storage using the `sceScreamValidateFileTokenStorage()` function, and retrieve individual file tokens using the `sceScreamGetFileTokenFromStorage()` function.

Validating a File Token Storage

Before validating a previously created file token storage, you must first load it into application memory. You can then call the `sceScreamValidateFileTokenStorage()` function, identifying the storage by a pointer to its location in memory. Because the `sceScreamValidateFileTokenStorage()` function may modify the file token storage, it should reside in read-write memory.

Retrieving a File Token

Before retrieving a file token from a file token storage you must first validate the storage in application memory. See above.

You retrieve a file token from a file token storage using the `sceScreamGetFileTokenFromStorage()` function. In the function's *`storage`* parameter you identify the storage from which to retrieve a token by a pointer to its location in memory. Using a zero-based index you also specify a file token to retrieve in the function's *`tokenIndex`* parameter.

> **Note:** The index used for the `sceScreamGetFileTokenFromStorage()` function's *`tokenIndex`* parameter must match that of the `SceScreamSndStreamFileToken` array that was passed to the `sceScreamCreateFileTokenStorage()` function's *`tokens`* parameter when the file token storage was created.

Finally, in the function's *`parseParams`* parameter, you point to the `SceScreamSndStreamParseParams` structure that was initialized as input to the `sceScreamParseStreamFile()` function when the specified file token was created. This `SceScreamSndStreamParseParams` structure contains a path pointer (in its *`file`* member) that is used to reopen the Stream file in asynchronous mode when streaming.

> **Note:** The `SceScreamSndStreamParseParams` associated with a Stream file must remain valid for the life of a corresponding file token.

> **Note:** A file token is an opaque pointer into an associated file token storage. So if you delete a file token storage, the tokens it contains become invalid.

10 Working with Multi-Layer Streams

This chapter explains how to work with multi-Layer Streams.

Understanding Multi-Layer, Multi-Bitstream Stream Files

The Sndstream library supports Stream files containing multiple Layers and interleaved Bitstreams. A Layer is a grouping of one or more Bitstreams that can be individually addressed and manipulated. A Bitstream is a strand of audio data that can be played on a single synthesizer voice. When multi-Bitstream files are compiled, the respective data from each Bitstream is interleaved into file contents. This enables simultaneous access to data from all Bitstreams using a single read operation.

The interleaved Bitstream mechanism offers considerable efficiency advantages when streaming data from optical media (characterized by a relatively high data throughput, but slow seek times). And it provides sample-accurate synchronization of multiple Bitstreams, allowing you to manipulate multi-Layer Streams as if live-mixing a multi-track recording.

Creating Multi-Layer Stream Files

You can create multi-Layer Stream files using Stream Creator, a component of Scream Tool. For information on doing this, see the "Stream Creator" chapter of *Scream Tool Help*.

Note: Stream Creator accepts input files in the `WAV` codec only. Output multi-Layer Stream files have an `XVAG` extension.

Manipulating Layers Individually or Collectively

Calls to `sceScreamStartStream()`, or other Stream-starting functions, that reference a multi-Layer Stream file return a Stream handle, just like a standard Stream file. However, you can also retrieve handles for individual Layers. And you can use these handles with the Scream `sceScreamSetSoundParamsEx()` function to set any parameter, including synthesizer parameters, on an individual Layer. You can also use the Scream `sceScreamAutoGain()` and `sceScreamAutoPan()` functions to automate incremental updates to gain or pan parameters on individual Layers. For further details about these functions, see the *Scream Library Reference* documents.

Figure 4 depicts a standard (single-Layer, single Bitstream) Stream. This is contrasted with a multi-Layer, multi-Bitstream Stream shown in Figure 5.

Figure 4 Standard (Single-Layer/Bitstream) Stream

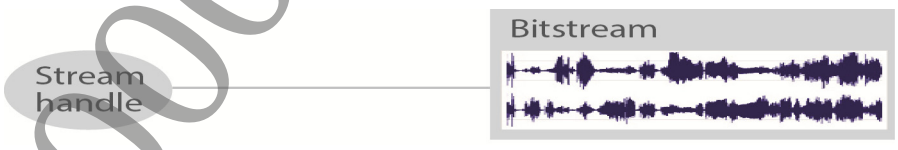
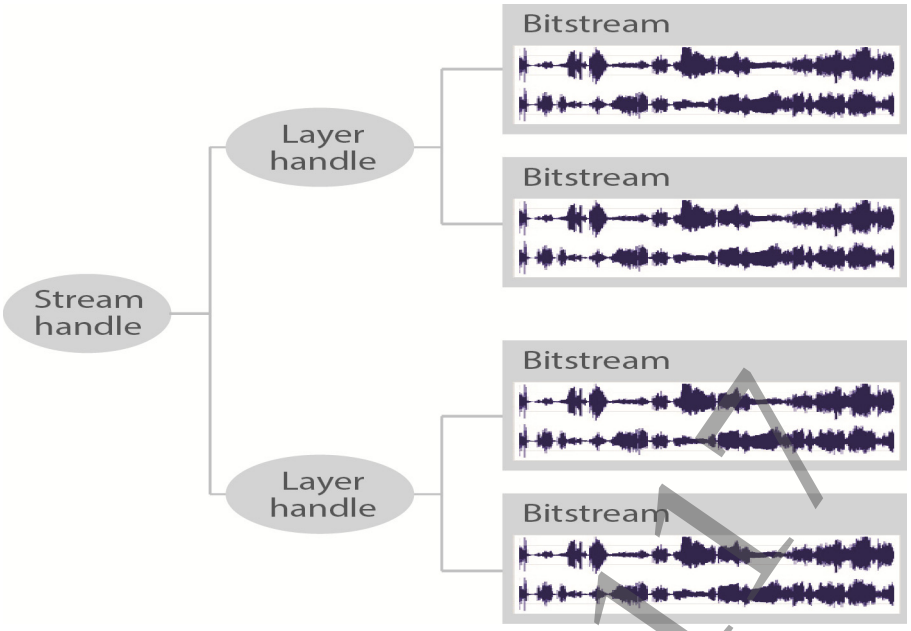


Figure 5 Multi-Layer/Bitstream Stream



As well as manipulating Layers individually, you can also collectively set and retrieve parameter values, and automate gain and pan changes, on all Layers in a multi-Layer Stream. For further details, see Setting Layer Parameters Collectively in the "Manipulating an Active Stream" chapter.

Constraints on the Number of Bitstreams

The maximum number of Bitstreams that can be associated with a Stream handle is 16 (defined by the Sndstream system constant `SCE_SCREAM_SND_STREAM_MAX_BITSTREAMS`). Note however, that the number of Bitstreams also cannot exceed the number of system-wide Stream handles, a value set when initializing Sndstream in the `sceScreamInitStreaming()` function's *`handleCount`* parameter. The initialized *`handleCount`* value can therefore further constrain the number of system-wide Bitstreams.

Retrieving Individual Layer Handles

You can retrieve individual Layer handles from a multi-Layer Stream file. You can then use these handles with the Scream `sceScreamSetSoundParamsEx()` function to set any parameter, including synthesizer parameters, on an individual Layer. You can also use Layer handles with the Scream `sceScreamAutoGain()` and `sceScreamAutoPan()` functions to automate gain and pan parameters on an individual Layer.

To retrieve individual Layer handles you must first determine the number of respective Layers associated with a Stream. You can do this either by reference to a Stream handle or file token. The respective functions are as follows:

- `sceScreamGetStreamLayerCountByHandle()`
  Retrieves the number of Layers associated with a Stream by handle reference.
- `sceScreamGetStreamLayerCountByFileToken()`
  Retrieves the number of Layers associated with a Stream by file token reference.

Having retrieved the number of Layers associated with a Stream, you can retrieve individual handles using the `sceScreamGetStreamLayerHandle()` function. You specify the handle of the containing Stream, and a target Layer for which to retrieve a Layer handle, using a zero-based index.

> **Note:** Layer handles are only retrievable from multi-Layer Stream files. If queried on a standard Stream file, the Stream handle is returned.

For a demonstration of manipulating Stream Layers, see the Sndstream Play Info sample program in the "Introduction" chapter.

Multi-Layer/Bitstream Files and the Asynchronous File I/O Functions

The asynchronous file I/O prototypes include a `bitstreamId` parameter that is used to identify a target Bitstream for file access purposes. When Sndstream calls your custom asynchronous file I/O functions, however, the values it uses for `bitstreamId` vary according to the type of operation. For efficiency purposes regarding files containing multiple Bitstreams, because Bitstream data is contained within the same file, Sndstream selects one `bitstreamId` value to serve as a reference for all file open and file close operations. Only for read operations does Sndstream specify the precise Bitstream for which it requires data.

For further details about the custom file I/O prototypes, see Using Custom File I/O Functions in the "Working with System Globals" chapter.

Supported Audio Formats

Supported audio data formats (codec and number of channels) for multi-Layer Stream files vary according to rendering synthesizer. Format specifications are shown in Table 5.

Table 5 Supported Audio Formats for Multi-Layer Stream Files

Synthesizer	ATRAC9™ Codec	VAG/VAG-HE Codec
NGS2	1, 2, 6, or 8 channels	1 or 2 channels
NGS	1 or 2 channels	1 or 2 channels

Multi-Layer Stream files must also meet the following criteria:

- same number of Bitstreams in each Layer
- same channel count in each Bitstream
- same sample rate for each Bitstream
- same codec used for each Bitstream

Further, multi-Layer Stream files, queued to the same Stream handle, must contain the same number of Bitstreams.