NGS Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

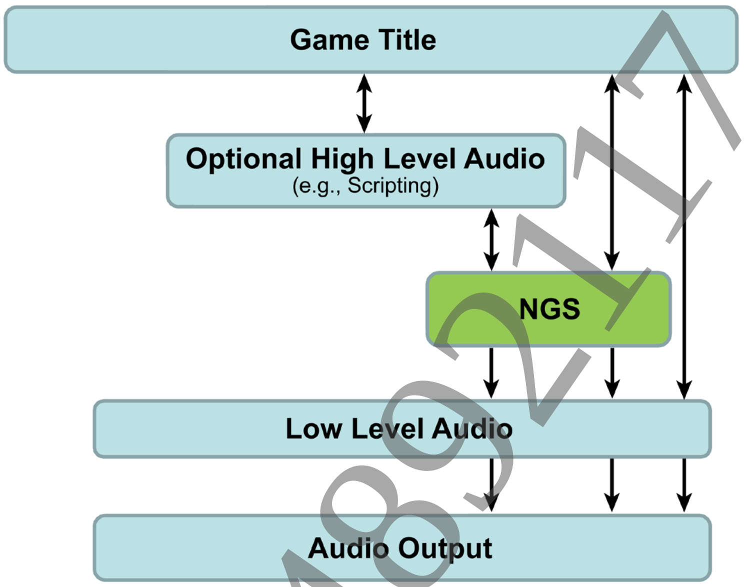
Table of Contents

1 NGS Overview

Introduction

NGS is an audio engine designed for games; it is designed around modular principles and allows users to play back multiple sources at independent frequencies. It allows simultaneous decoding of multiple formats, contains powerful DSP algorithms, and allows configurable, flexible routing.

Figure 1 Audio Subsystem Interaction



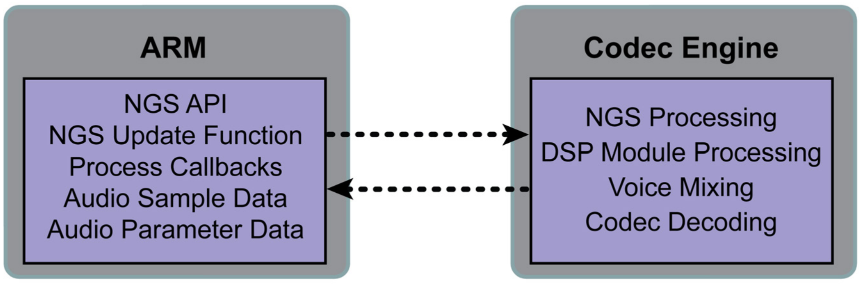
Codec Engine and ARM Processing

NGS processing takes place in two areas within the PlayStation®Vita hardware. The API side (i.e., processing any requests by the user, or feeding back the status of NGS to the user) is processed by the ARM processor. The actual processing of audio data (resampling, codec decoding, mixing and applying digital signal processing (DSP) effects, such as filters or distortion) is handled by a separate hardware unit, called the Codec Engine.

The Codec Engine hardware allows NGS to process audio data in parallel to the ARM processor, therefore reducing CPU load on game applications.

Note: It is not possible for users to access the Codec Engine for their own use.

Figure 2 NGS Processing



Files

The files required to use the NGS API are listed in Table 1.

Table 1 Required Files

File Name	Description
`ngs.h`	Main header file
`ngs/ngs_top.h`	System API header file
`ngs/error.h`	Error return code header file
`ngs/modules/*.h`	Module header files
`ngs/templates/*.h`	Voice template header files
`sulpha_ngs.h`	Debugger header file

Note: sysmodule is required to startup NGS. For more information, refer to the *libsysmodule Overview* and *libsysmodule Reference* documents.

Supported Audio Formats

Audio data for NGS is either supplied to NGS in a buffer, or created by NGS using the Signal Generator Module.

For data supplied in a buffer, the following formats are supported:

- PCM Audio: Player Module
- VAG: Player Module
- HE-VAG: Player Module
- ATRAC9™: ATRAC9™ Module

Note: NGS does not support ATRAC9™ Band Extension, the extended format of ATRAC9™. You can determine the format from the value of `dwVersionInfo` in the `RIFF` Header. For more details, refer to the *ATRAC9™ File Format* document.

Tools are provided with the SDK for creating and viewing data in the above formats. These tools can be found in the `%SCE_PSP2_SDK_DIR%\host_tools\bin` directory.

- at9tool: For encoding and decoding ATRAC9™ files
- VAG Converter 2: For encoding VAG/HE-VAG files

These tools are described in the following documents:

- *at9tool User's Guide*
- *VAG Converter 2 Tools User's Guide*

Debugging NGS-Based Audio

NGS provides a wide and complex range of features to support your audio needs. The Sulpha tool is provided to aid with debugging your NGS-based audio system and is particularly useful for determining the cause of audio glitches and audio routing issues.

Sulpha is documented in the *Sulpha Tool User's Guide*.

Basic Playback

To play back basic PCM audio data, the following steps must be performed:

(1) Initialize NGS.

(2) Create a master-buss rack with 1 voice.

(3) Create a rack of voices with a Player voice.

(4) Route a Player voice to the master-buss voice by creating a Patch.
(5) Load some PCM audio data into a memory buffer.
(6) Assign the PCM memory buffer to the Player voice.
(7) Play the master-buss voice and the Player voice.

To process and output audio to the hardware device, the following steps must be performed:

(1) Call `sceNgsSystemUpdate()`.
(2) Obtain the output buffer from the master voice using `sceNgsVoiceGetStateData()`.
(3) Pass the contents of the output buffer to `libAudio` via `sceAudioOutOutput()`.

The processing is normally executed in a separate thread.

These steps are shown in the `pcm_player` sample code.

Reference Materials

For detailed information, refer to the following documents:

- *NGS Reference*
- *NGS Modules Reference*
- *NGS Modules Overview*

2 Terminology

The following list defines the terminology used throughout NGS:

System

The main system object for the NGS instance.

Module

Modules are self-contained subsystems that process data, such as decoders, DSP effects, and mixers. They are plug-in based and can handle 1 to n channels.

Params

Params are parameters, which can be modified by the user to affect processing. They are the method by which the user passes data to Modules.

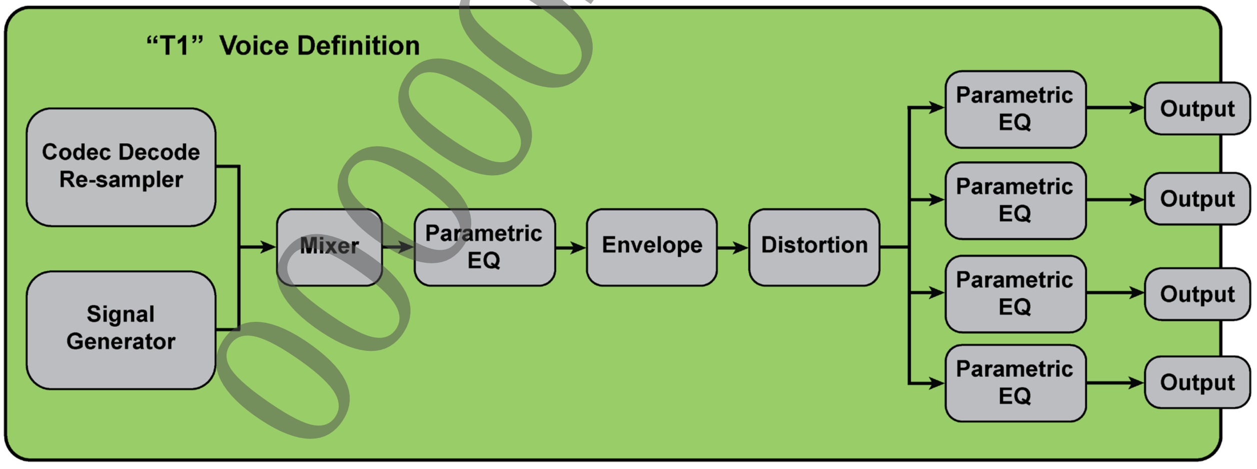
States

States refer to the current state of a Rack, Voice or Module. State data comes in two formats: system state data and user state data. The user can query a module for its user state data.

Voice Definition

A Voice Definition is a data structure representing one or more Modules, their interconnectivity and processing order. A Voice Definition may also contain Voice Presets (see Figure 3).

Figure 3 Voice Definitions and Modules



A Voice Definition contains modules.

Modules can be bypassed if desired.

Voice

A Voice is an active instance of a Voice Definition. It can be controlled by the user to play, stop, pause, etc. Voices can either be generators, for example a decoder voice, or act as data processors, for example a reverb.

Voice Preset

A Voice Preset is configuration data for a Voice. It contains Params and/or bypass flags for some or all of the modules within a Voice.

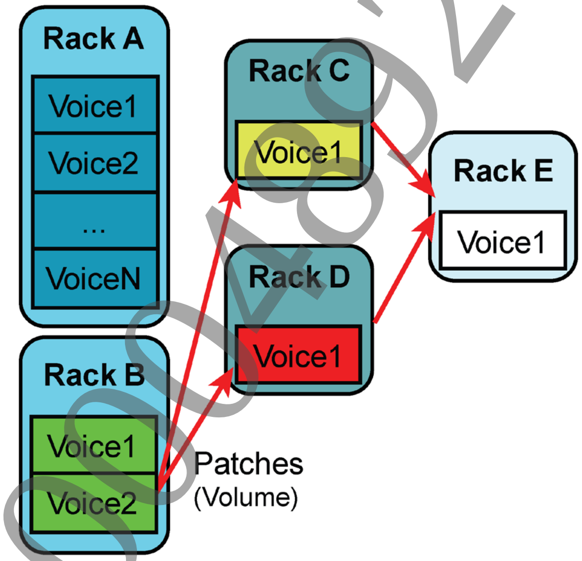
Template

A Template is a fixed Voice Definition, supplied additionally with NGS.

Rack

A Rack is a container unit for 1 to n Voice instances as defined by a single Voice Definition. The Rack defines the number of audio channels, as well as the number of input and output Patches per Voice instance.

Figure 4 Racks, Voices and Patches



Each Rack contains 1-n voices.

Each Voice within a Rack uses the same Voice Definition.

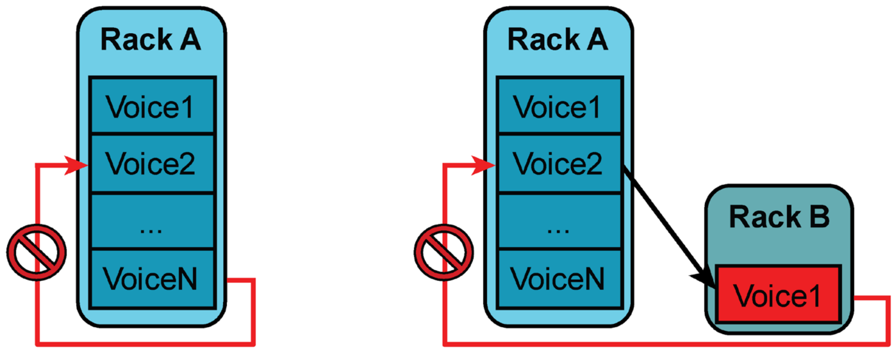
Voices are connected to other voices via patches.

Patch

A Patch is a connection between two Voices (a source and destination). The Patch contains the volume information for the specified routing.

Note: Creation of patches which would cause circular dependencies between Voices or Racks will fail.

Figure 5 Invalid Patch Connections



Voices cannot connect to another voice within the same rack.

Voices cannot connect to another rack in a way which would result in a circular dependency.

Buffer

A Buffer represents a group of PCM audio channels. The size of a Buffer depends on the granularity of the System.

3 Codec Support

NGS can support the following codecs via its Player Modules:

- PCM (16 bit, mono or stereo, 0-192KHz)
- VAG
- HE-VAG (High quality VAG)
- ATRAC9™

Refer to the relevant documentation regarding these codecs for further information.

NGS uses two different Voice Definitions to allow the user to process codecs:

- VAG / HE-VAG and PCM support
  - `voice_template_1.h`
  - `simple_voice.h`
  - `sas_emu_voice.h`
- ATRAC9™ support
  - `voice_template_at9.h`

See the Voice Definition Template section for further information regarding Voice Definitions.

Codec Encoding Tools

To create VAG or HE-VAG files, use the Windows VAG Converter 2 tool, which is installed as part of the PlayStation®Vita SDK in `%SCE_PSP2_SDK_DIR%\host_tools\bin`.

To create ATRAC9™ files, use the Windows at9tool, which is installed as part of the PlayStation®Vita SDK in `%SCE_PSP2_SDK_DIR%\host_tools\bin`.

See the SDK documentation for further details regarding these tools.

4 NGS Operational Detail

The System instance, which must be pumped (by an update thread for example) to create audio output packets through calls to its update function, acts as a container for one or more Racks. At any time during the lifetime of the System you can create or destroy a Rack. The System functions enable you to synchronize singular or multiple Param changes to Voices within the System Racks.

In the same way that the System acts as a container for Racks, a Rack acts as a container for one or more Voices of identical topology (in Figure 6 the Voices in Rack 0 all have the same topology). A single Rack will process all the Voices contained within it before the process moves on to the next Rack in the sequence.

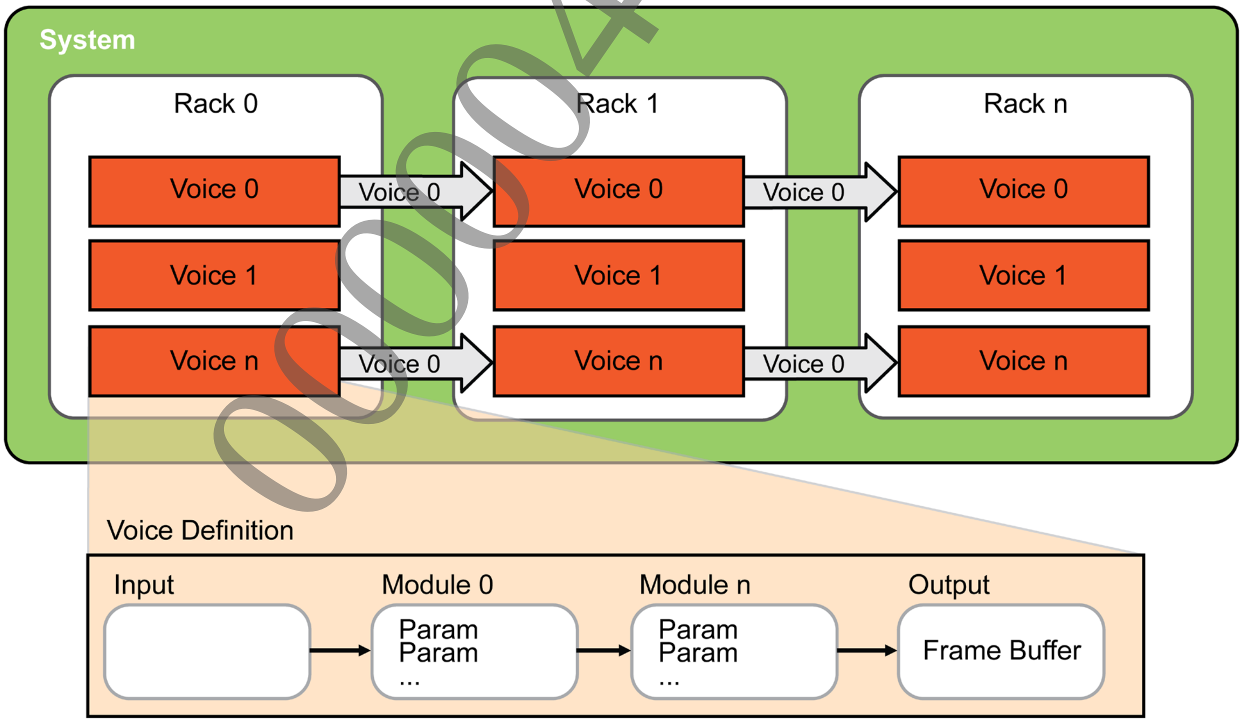
Voices can be either generators (those that handle sample playback) or processors (those that define a reverb buss). In both cases, the audio data in a Voice is processed through an internal graph of Modules; the topology of the graph is defined in a Voice Definition.

Figure 6 shows the Voice Definition for the Voices in Rack 0. Each Voice Definition contains an input. This could be either PCM data or from a Signal Generator. The data sequentially cascades through a defined set of Modules and outputs into a Buffer, which can be used as the input for a Voice in the next Rack. The routing of audio data between Voices is handled at runtime via Patches. These patches control both the unidirectional flow and scaling of audio data.

Voice Definitions can be applied through templates, keeping the Module setup for each voice the same. The template is purely a placeholder for each Module; it does not dictate what that Module will do, if anything, but acts as a starting point for initializing Voices.

Each Module has a set of parameters; it is these that make a generic Voice Definition Template into something specific. For more information, see Setting Parameters.

Figure 6 Architectural Overview



Using Voice Presets

When initializing voices, it is likely that you will want to apply common settings to the modules of that voice. To minimize the amount of code required to initialize a voice, user-defined presets can be applied during voice initialization.

A voice preset is defined using the `SceNgsVoicePreset` structure. This structure describes the initial module parameters and which modules should be bypassed.

The preset data requires a contiguous memory block of alternating `SceNgsModuleParamHeader` and parameter buffer structures. These structures are already defined contiguously in the block format for the parameters. For example, to create user-defined preset data for a Player Module and Parametric EQ, you could create the following structure in your application code:

```
typedef struct UserPreset
{
    SceNgsPlayerParamsBlock  playParams;
    SceNgsParamEqParamsBlock eqParams;
} UserPreset;
```

The list of modules to bypass is simply an array of module IDs.

Voice State Transitions

The following diagrams show the transformations which can be applied to Voices during their life cycle.

Figure 7 Voice State Transition Graph

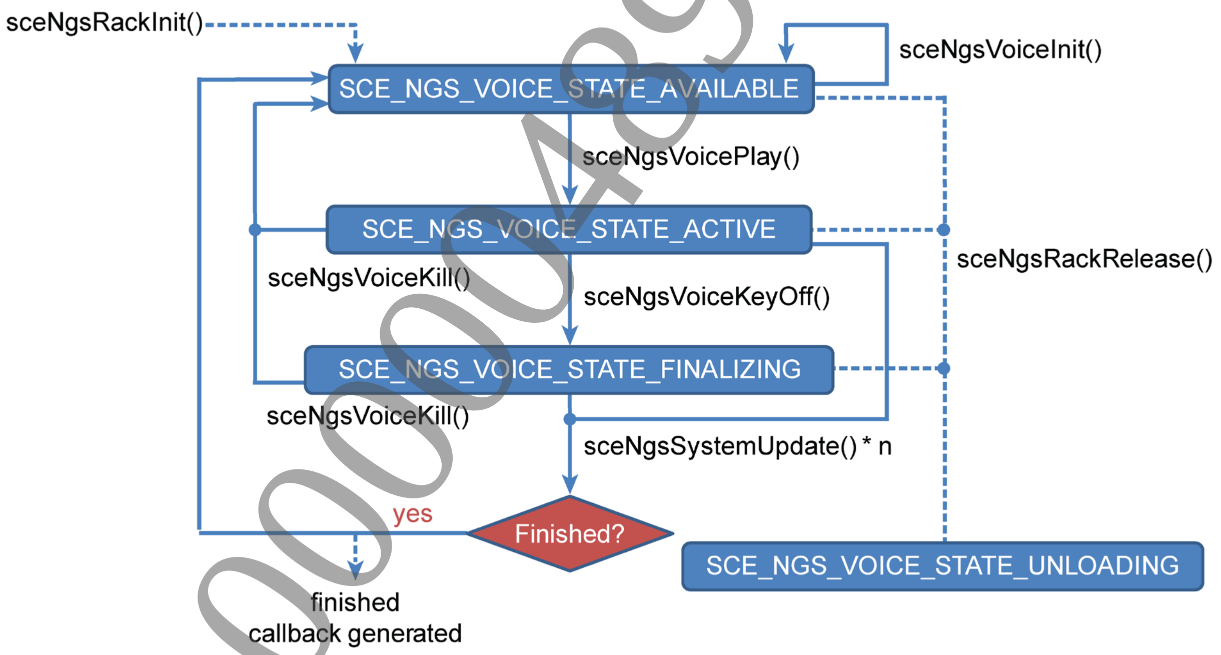
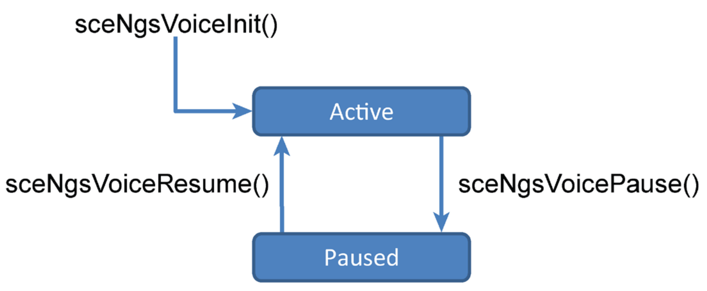


Figure 8 Voice Pause Transition Graph



It is possible to query the state using the function `sceNgsVoiceGetInfo()` and examine the *`uVoiceState`* field.

> **Note:** To receive a finished callback from the system, the voice must either go through the finalizing state `sceNgsVoiceKeyOff()` or have finished playback of a non-looping sound. The callback will not be generated if the voice is immediately stopped via `sceNgsVoiceKill()`. See `sceNgsVoiceSetFinishedCallback()` for details of setting up a callback.

The main states can be defined as shown in Table 2:

Table 2 Voice Main States

State	Description
SCE_NGS_VOICE_STATE_AVAILABLE	Voice is not actively cued to play.
SCE_NGS_VOICE_STATE_ACTIVE	Voice is actively cued to play or playing.
SCE_NGS_VOICE_STATE_FINALIZING	Voice is actively cued to play or playing, but is in a release phase.
SCE_NGS_VOICE_STATE_UNLOADING	Voice is in a rack which is or has been released and is no longer accessible.

The flags listed in Table 3 can also be logically ORed into the value:

Table 3 Voice Flags

Flag	Description
SCE_NGS_VOICE_STATE_PENDING	Voice has received a note-on request but is not yet in the active state.
SCE_NGS_VOICE_STATE_PAUSED	Voice is in a paused state.
SCE_NGS_VOICE_STATE_KEY_OFF	Voice has received a note-off request but is not yet in the finalizing state.

5 Example NGS Configuration

Figure 9 shows an example configuration of a Patch and Rack setup which could be used for a game title utilizing two source Racks.

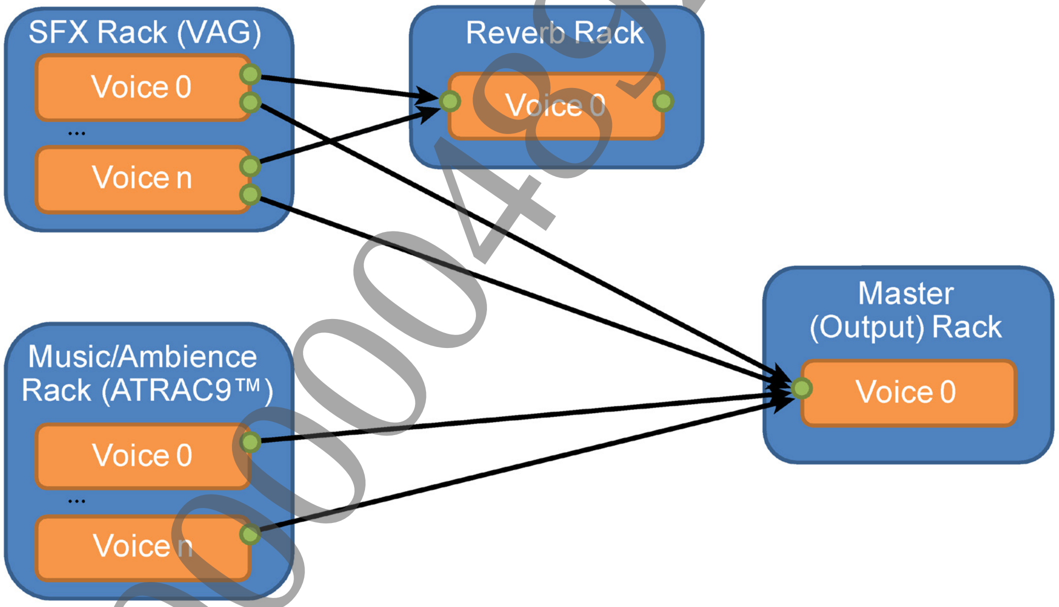
The Master (Output) Rack is used to handle final mixing and return the generated PCM to the user via calls to `sceNgsVoiceGetStateData()`.

The Reverb Rack is used to apply a reverberation effect to the game sound effects to simulate room acoustics.

The Music/Ambience Rack is used for playback of music and background ambience sound files stored in the ATRAC9™ format. Typically music and ambience files are longer in duration and therefore are saved in a more highly compressed format which trades smaller file size with greater decoding overhead.

The other source Rack is used for the game sound effects (SFX) which are VAG, HE-VAG, or PCM source files. The SFX rack has two outputs per Voice. One is connected directly to the master output Voice and is used for the 'dry' signal; the other is connected to a reverb Voice (the 'wet' signal). It should be noted that it is possible to connect multiple patches from a single output (see `SceNgsPatchSetupInfo`, `nSourceOutputSubIndex`). The advantage of using separate outputs is that both can be processed differently within the voice. In the example, each output can be filtered with separate filter settings to simulate obstruction and occlusion effects within the game environment.

Figure 9 Example Configuration

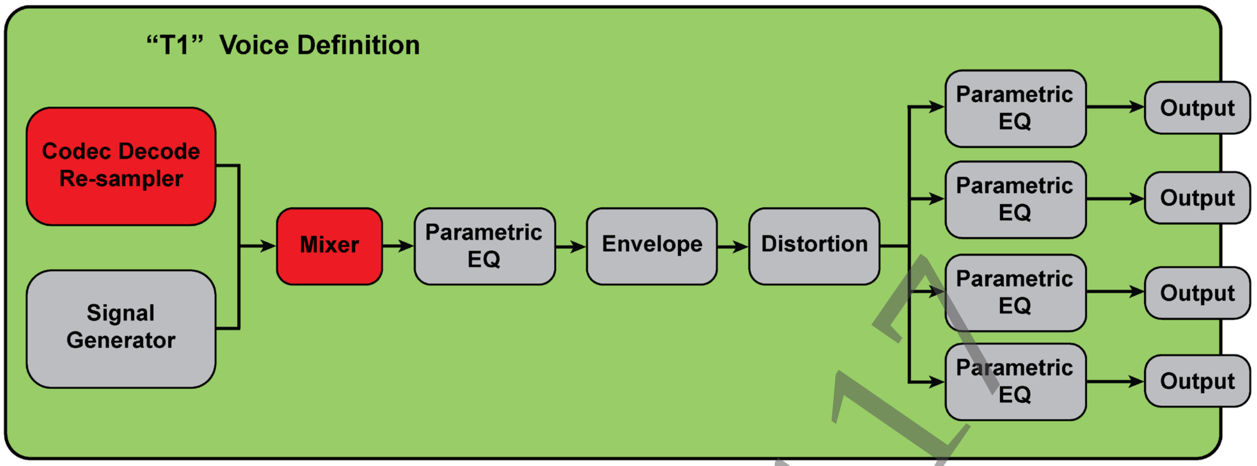


Voice Definition Template

Voice Definition Templates can be used to initialize the System quickly. Voice Definitions tell the System which modules to use and how they are to be connected. Essentially, this creates a fixed path architecture. Voice Definition Templates allow for a modular System to be set up very simply. They contain all of the information for Voice, signal routing, required modules (DSP effects, for example) and so on.

Various Voice Definition Templates are supplied with NGS. You can use these to initialize the Racks and create a unique audio system. The Voice Definition Template is designed as a general purpose generator Voice, which can handle most game case voices. It contains both signal generation as well as data re-sampling playback, four separate outputs, each with the option of independent filters as well as various other modules (distortion, envelope, etc.).

Figure 10 Voice Definition Template Default Bypass Status



By default the red blocks in Figure 10 (representing Modules) are active and the grey blocks are bypassed. Therefore, when using this template, the only Modules active by default are the Codec Decode and the Mixer.

Note: There are two voice templates for this arrangement, with differing Codec Decodes: one for ATRAC9™ audio data; one for PCM/ADPCM (VAG or HE-VAG) audio data.

Specific module templates are located in the `ngs/templates` directory. Every template found in this directory corresponds to the `sceNgsVoiceDefGet*` function for that module, and the template file must be referenced when using the command.

For example, `sceNgsVoiceDefGetMasterBuss()` requires that you have included the `ngs\templates\master_buss_voice.h` file.

Other templates have an input mixer module, a DSP module (reverb, delay etc.) or a PCM output module.

Setting Parameters

Each DSP Module has specific attributes that change its behavior and settings. There are two methods through which you can interact with the Param (parameter) data of the Modules. Each method is suited to different requirements and it is recommended that you follow the suggestions to ensure the most optimal performance from the System.

(1) Method A: Block Parameter Setting

This method allows you to batch together changes to one or multiple Modules within a Voice and pass the changes via a single instruction call. This would generally be the preferred method in situations such as the initialization of a Voice where all parameters of each specified Module must be initialized.

To use this method, you need to use the Modules parameter block structure that includes header information and Module identification.

(2) Method B: Lock / Unlock

This method allows you to update a single parameter at runtime. You user must lock and unlock parameter interfaces for each Module within a Voice before setting the parameter value. Changes to locked Param interfaces will not be applied until the interface has been unlocked.

For more information on parameter settings and how to implement the two methods, see the *NGS Modules Reference*.

Some NGS modules define parameter presets to avoid the need to set common groups of parameters. For example, the Reverb module lists a range of different reverb types that can be used directly to avoid specifying the individual low level parameters.

Presets can be used in the following way:

(1) Lock the module parameters using `sceNgsVoiceLockParams()`.

(2) Pass the parameter buffer from `sceNgsVoiceLockParams()` to `sceNgsModuleGetPreset()`. This will fill the buffer with preset values.

(3) Optionally modify any values of the preset.

(4) Unlock the parameter buffer using `sceNgsVoiceUnlockParams()`.

System Caveats

Patch Connections

- Although branching and merging is allowed by patching, when Patches are created it is not permitted to create circular dependencies between Voices; NGS will not allow you to create such a setup (see Figure 5).
- It is not permitted for you to create Patches in which the input and output Voice exist within the same Rack (see Figure 5).

Module Runtime Modifications

- The internal processing graph within a Voice, such as Module-to-Module routing, is fixed upon creation of the Rack and may not be edited at runtime. The only runtime modifications allowed per module are bypassing of modules and Param settings.

NGS Granularity

- A packet of PCM data is output when NGS processes. The size of this packet is known as the *granularity* per channel (see `SceNgsSystemInitParams`). Sizes of 64, 128, 256 and 512 are valid.

Memory Restrictions

- NGS internally has a finite, non-exposed memory pool used by the Codec Engine. When the NGS System and Racks are initialised, this allocates memory from within this pool based on the Voice Definition, module types, number of voices requested etc., with some modules (such as the Delay and Reverb) using larger amounts of memory than others. When this pool is exhausted or the requested allocation is too large, the user will receive a `SCE_NGS_ERROR_OUT_OF_ASSETS` return code from the initialisation functions.

6 Streaming

To stream audio in NGS it is required to use two or more buffers for the player modules (see "ATRAC9™ Player DSP Effect Module" and "Player DSP Effect Module" in the *NGS Modules Reference*) and set up a callback for the module using `sceNgsVoiceSetModuleCallback()`.

Each module will generate a maximum of 1 callback per update frame, this being the last generated callback. The user should ensure that when streaming, the buffers are large enough so that only a maximum of 1 buffer will be consumed each update; otherwise there is a risk that the audio data will be consumed faster than the buffers can be filled.

All callbacks are serviced post processing in the call to `sceNgsSystemUpdate()` and only the last generated callback from the module is reported in the frame. Therefore, if the buffers are too small it is possible that NGS may generate multiple module callbacks, but the callback is only returned once, with the last set of data.

A simple double buffered handling callback would operate in the following way. Before starting the voice the user should ensure that both buffers are filled. Then in the callback, fill the buffer which has just finished playback; for example:

```
void playerCallback( const SceNgsCallbackInfo *pCallbackInfo )
{
    if(pCallbackInfo->nCallbackData == SCE_NGS_PLAYER_SWAPPED_BUFFER)
    {
        int nLastFinishedBuffer = pCallbackInfo->nCallbackData2;
        StreamDataIntoBuffer(pCallbackInfo->hVoiceHandle,
                             nLastFinishedBuffer);
    }
}
```

The size of buffers required is dependent on various factors such as the compression rate or encoded format (or both), the number of channels, the system granularity, the playback frequency, and any start or end offsets supplied.

The following suggested guidelines for minimum buffer sizes for each streaming buffer (i.e., per side in a double buffered system) are based on a system granularity of 512 samples, maximum playback rates as shown, and assuming no start or end offsets are supplied.

Note: Larger buffers than those suggested may be preferable as you would need to service them less frequently and therefore would get a better overall system performance (for example, fewer large load or copies, versus more small load or copies).

Table 4 Streaming Buffer Minimum Buffer Size Guidelines

Format	Maximum Playback Rate	Size (bytes)
Mono PCM	48 kHz	2048
Stereo PCM	48 kHz	4096
Mono VAG	48 kHz	512
Stereo VAG	48 kHz	1024
ATRAC9™ 96 kbps	48 kHz	256
ATRAC9™ 192 kbps	48 kHz	512

When using these guidelines you should modify the sizes based on the following formulae:

size * = (max playback rate / 48 kHz)

size * = (NGS system granularity / 512)

size + = (start offset bytes / compression rate)

size + = (discard end bytes / compression rate)