Audio Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Overview

The contents of document

This document provides an overview of 5 types of sound libraries prepared for PlayStation®Vita SDK and describes how to use NGS, one of the sound libraries, that plays a key role when creating a sound in a game application.

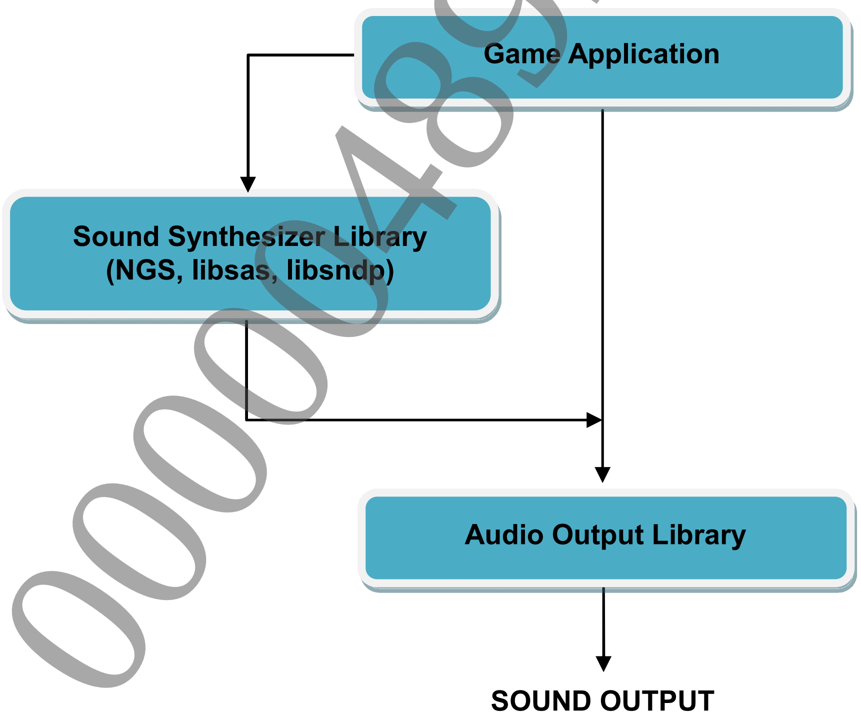
Sound Output Flow of PlayStation®Vita

The PlayStation®Vita SDK provides the following sound libraries.

- Audio output library (low-level audio library)
- NGS (sound synthesizer library)
- libsas (sound synthesizer library)
- libsndp (high level (MIDI playback) library)
- Scream (high level library)

A game application outputs sounds using these libraries. The sound output flow of PlayStation®Vita is as shown below.

Figure 1 Sound output flow



2 Overview of Sound Library for PlayStation®Vita

The 5 libraries introduced in chapter 1 are described below.

You are recommended to familiarize yourself with these features when building a sound system so as to implement settings appropriate for each game application and routing scenario.

Audio Output

This is a low level audio library for outputting audio data in the PCM format (16bit LPCM, 2chstereo or monaural).

There are 3 ports such as the MAIN port, the BGM port and the VOICE port which can be used separately depending on the features. The final audio output will have 3 ports mixed.

As for details, please refer to the following documents.

- "Audio Output Function Overview"
- "Audio Output Function Reference"

NGS

This is a sound synthesizer library for PlayStation®Vita that works on the Codec Engine, not the main CPU.

It enables you to design voices (refer to the NGS High Level Concept chapter) in details such as the routing between voices, thus a sound system appropriate to each game application can be created. Also, various effects such as the reverb and filters are provided.

Note that this library is for creating wave format data, thus it is required to use the audio output library to output the created audio.

As for details, please refer to the following documents.

- "NGS Overview"
- "NGS Reference"
- "NGS Modules Overview"
- "NGS Modules Reference"

libsas

This is a simple and easy-to-use sound synthesizer library that works on the main CPU.

The compatibility with the libsas in PSP™ (PlayStation®Portable) SDK is kept by extending some features for PlayStation®Vita such as the number of voices.

Note that this library is for creating wave format data, thus it is required to use the audio output library to output the created audio.

As for details, please refer to the following documents.

- "SAS Overview"
- "SAS Reference"

libsndp

This is a high level library that works on libsas and prepared for keeping the compatibility with PSP™, thus no future extension will be arranged.

To create sound data that works with this library, use the sndConv package that is available from the PSP™ Developer Network (https://psp.scedev.net).

Thus, when newly using the sound libraries for PlayStation®Vita, use the NGS, libsas or Scream sound libraries or such. Note that this library is for creating wave format data, thus it is required to use the audio output library to output the created audio.

As for details, please refer to the following documents.

- "libsndp Overview"
- "libsndp Reference"

Scream

This is a high level library that works on NGS. Because it can make use of the features of NGS and supports script playback and the audio streaming feature, it allows you to easily create and use effects as well as various sounds. Scream Tool is also provided as a tool for creating data assets for Scream.

Because multiple libraries are provided as described above, the diverse needs such as "to port data from PSP™ easily" or "to develop more complex design" can be met and you can select and use a specific sound library depending on the game you develop.

Note that this tutorial focuses on NGS, which works on the Codec Engine and requires detailed settings.

3 NGS High Level Concept

NGS builds a sound system using a module, voice, rack and patch.

You are recommended to get familiar with these features when building a sound system so as to implement detailed settings according to each game application and perform routing.

Module

A module is a self-contained subsystem that handles data such as the Decoder, DSP effect and Mixer. These modules are provided as plug-ins and can handle 1 to "n" number of channels.

Such modules are PCM Player, Mixer, EQ (Equalizer) in the template1 voice example (provided as a voice template in NGS) as shown in the Figure 2. A voice is made up of a combination of these modules. Also, setting these modules in details allows you to finely control the EQ.

Also, the user can bypass these modules and use their parameters to fine control the sort of settings one would normally expect to find in an audio system.

As for further details, please refer to the following documents.

- "NGS Modules Overview"
- "NGS Modules Reference"

Voice

A voice is composed of any number of modules and can act as generators, producing sound, or act as data processors, adding effects and the like. In the following sections we look at an example of each.

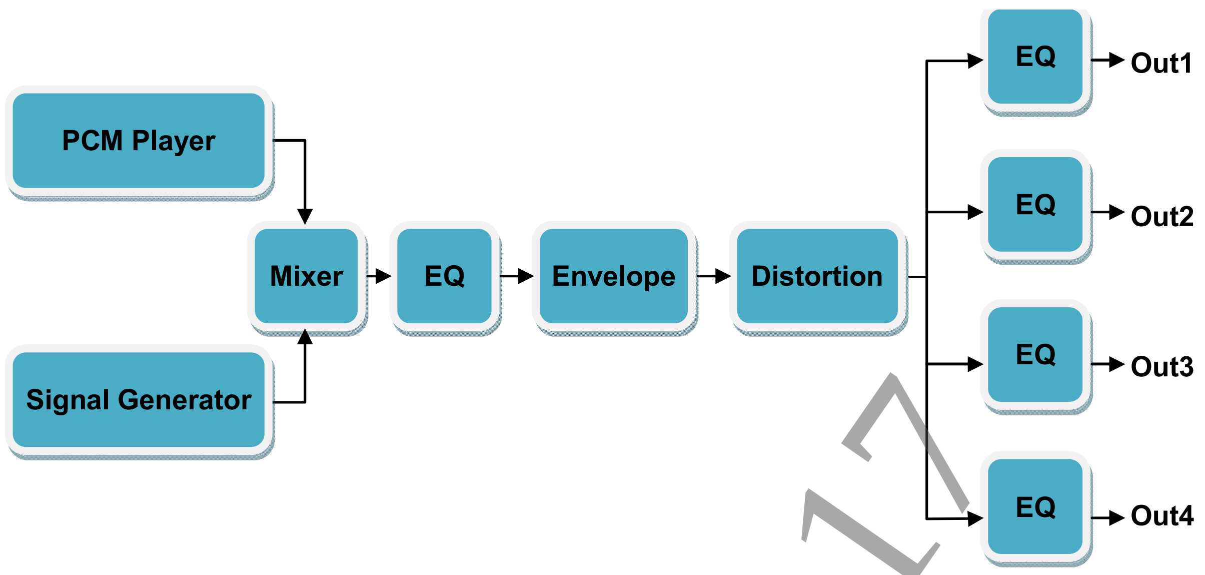
An Example of a Generator Voice

Like the template1 voice as illustrated in Figure 2, a voice that generates audio based on the PCM data or from the signal generator is treated as a voice for the generator.

Its characteristic is described below.

- Possible to playback, stop and perform other operations of audio by controlling this type of voice
- By controlling the module parameter in the voice template like the one illustrated in Figure 2, effect like envelope or EQ can be controlled.
- Supports the PCM, ADPCM (VAG, HE-VAG (This is High quality VAG file format. Backward compatible with original PlayStation® VAG format)) and ATRAC9™ formats

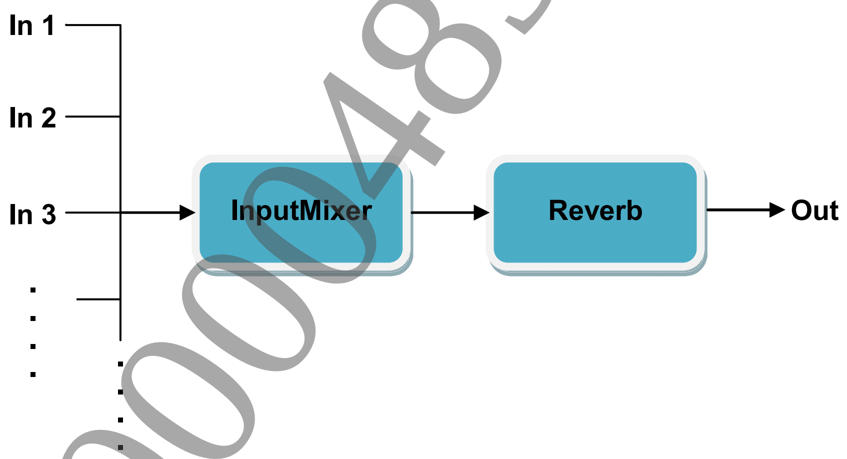
Figure 2 Structure of the template1 voice



An Example of a Data Processor Voice

The Reverb buss voice is illustrated in Figure 3. This voice handles other voices as an input/output and adds a reverb effect. Connects multiple voices to perform mixing or apply the reverb effect

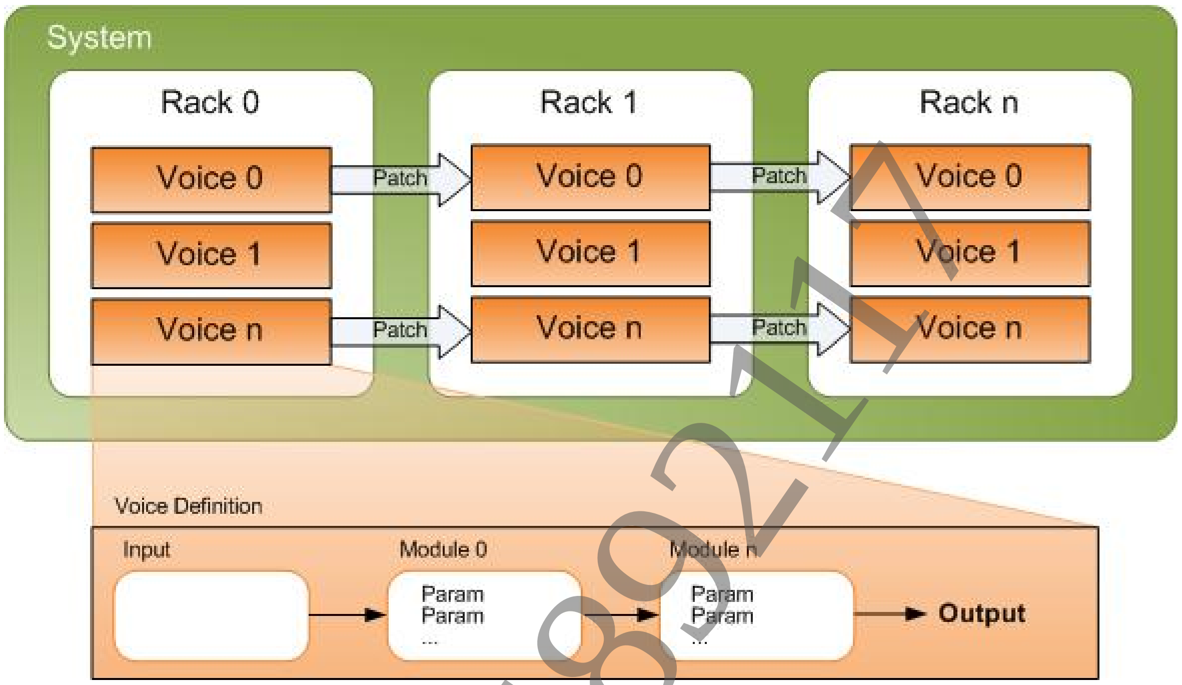
Figure 3 Structure of the reverb buss voice



Rack

As illustrated in the figure 4, the rack is a container unit that stores multiple voice instances. To run the system, it is required to create at least 1 rack. It is possible to consolidate voices per scene depending on game applications.

Figure 4 Overview of Module, Voice, Rack and Patch



Patch

As illustrated in Figure 4, the patch means the connection between 2 voices (source and destination). The patch contains volume data of the specified routing.

It is not possible to create a patch for voices within the same rack or a patch that causes circular dependencies between the voice and the rack.

4 Basic Usage of NGS

As described above, the basic usage of NGS using the module, voice, rack and patch will be explained in line with the pcm_player sample.

This sample sets only the parameters of the pcm player module, but how to set the parameters are common for all modules. Note that the purpose of each module and parameter differs. As for details, please refer to the "NGS Modules Overview" or the "NGS Modules Reference" document for more details.

pcm_player sample plays back audio as the following steps.

Note that this tutorial describes each step, but generation of the rack, voice, and patch, setting of parameters of a module or playback of the voice or such can be done at any time by the game application.

(1) Initialize the NGS system
(2) Generate a rack and a voice
(3) Set parameters of a module within the voice
(4) Generate a patch(connect voices)
(5) Play back the voice
(6) Update the NGS system
(7) Terminate the NGS system

Initializing the NGS system

Before initializing the NGS system, you have to first obtain/allocate necessary memory size for the NGS system.

Note that the memory to allocate should be 256 bytes aligned.

Example of initialization: api_libngs - initNGS

```
int initNGS(void)
{
    int  returnCode = SCE_OK;
    SceNgsSystemInitParams initParams;
    size_t     size;

    // Decide the value necessary for system initialization
    initParams.nMaxModules  = NUM_MODULES;
    initParams.nMaxRacks    = 2;
    initParams.nMaxVoices   = 2;
    initParams.nGranularity = SYS_GRANULARITY;
    initParams.nSampleRate  = SYS_SAMPLE_RATE;


    // Decide the memory size used by the NGS system
    returnCode = sceNgsSystemGetRequiredMemorySize( &initParams, &size );

    //Allocate the necessary memory for the NGS system
    s_pSysMem = _aligned_malloc( size, MEM_ALIGNMENT );

    //Initialize the NGS system and obtain the handle
    returnCode = sceNgsSystemInit( s_pSysMem, size, &initParams,
    &s_sysHandle );

    return returnCode;
}
```

Generating a rack and a voice

To generate a rack, you first need to define a voice to be stored in the rack. In the pcm_player sample provided in the following directory, the voice_template_1 and a voice for the master buss for mixing the final output of NGS are defined.

- %SCE_PSP2_SDK_DIR%\target\samples\sample_code\audio_video\api_libngs\pcm_player

In the sample, the template definition is used, however, each game application can define the format of the voice.

Note that the system provides a voice template that is frequently used by default. Normally, only this template suffices in most cases.

```
// Get voice definitions
pT1VoiceDef        = sceNgsVoiceDefGetTemplate1();
pMasterBussVoiceDef = sceNgsVoiceDefGetMasterBuss();
```

Secondly, set the parameters of the rack, allocate the memory and then initialize the rack. Note that the memory to allocate must be 256 bytes aligned.

Pass the previously defined voice definition here. If you wish to store multiple voices in the rack, increase the value specified to nVoices.

In this example, the voice definition voice_template_1 is passed and a rack that contains a single voice voice_template_1 is generated.

Note that the voices included in the same rack will have the same properties (number of channels, codec, DSP, etc.).

```
rackDesc.nChannelsPerVoice   = 2;
rackDesc.nVoices            = 1;
rackDesc.pVoiceDefn         = pT1VoiceDef;
rackDesc.nMaxPatchesPerInput = 0;
rackDesc.nPatchesPerOutput   = 1;

// Obtain the memory size necessary for generating the rack
returnCode = sceNgsRackGetRequiredMemorySize( s_sysHandle, &rackDesc,
&bufferInfo.size);

//Allocate the memory
s_pRackMemPlayer = _aligned_malloc( bufferInfo.size, MEM_ALIGNMENT );
bufferInfo.data = s_pRackMemPlayer;

//Initialize the rack
returnCode = sceNgsRackInit( s_sysHandle, &bufferInfo, &rackDesc,
&s_rackPlayer );
```

Setting parameters of a module within the voice

To set parameters of a module within the voice, you first have to obtain a handle of the voice from the rack.

```
//Obtain the handle of the voice from the rack
returnCode = sceNgsRackGetVoiceHandle( s_rackHandle, 0, &voiceHandle );
```

After that, set the module parameters using the handle.

Before setting the parameters, it is required to lock the module within the voice of which you want to set the parameters as described below.

```
//Lock the module
sceNgsVoiceLockParams( voiceHandle, //Handle of the voice
nModuleId,     //Module "Index" within the voice
SCE_NGS_PLAYER_PARAMS_STRUCT_ID,  //ID of the module of which parameters are set
&bufferInfo );
               .
               .
               .
/* Set the parameters here */
               .
               .
               .

// Unlock the module
sceNgsVoiceUnlockParams(voiceHandle, nModuleId);
```

The pcm_player sample sets only the player module, but parameters of other modules also can set in the same steps as shown below.

"Lock the module" -> "Set parameters" -> "Unlock the module"

Note that this tutorial shows only how to set parameters of a single module, however, by using `sceNgsVoiceSetParamsBlock()`, you can set parameters for multiple modules contained in a single voice at once.

Also, because it shows a one shot playback, the pcm player uses only a single buffer out of 4 buffers of the player module, however, using all of the 4 buffers allows you to easily perform file streaming play or loop playback in the middle of audio. For a detailed description of streaming playback of NGS, refer to the chapter 5 "Audio Playback with NGS".

Generating a patch(Connecting voices)

You first need to set/generate the `SceNgsPatchRouteInfo` structure for generating a patch as described below.

Note that the Pcm_player sample connects to the master buss, however, you can also set a connection to the reverb buss or such in order to apply the reverb effect. Please refer to the At9_Reverb sample which demonstrates how to use the reverb effect.

```
patchInfo.hVoiceSource          = hVoiceSource;
patchInfo.nSourceOutputIndex     = 0;
patchInfo.nSourceOutputSubIndex = SCE_NGS_VOICE_PATCH_AUTO_SUBINDEX;
patchInfo.hVoiceDestination     = hVoiceDest;
patchInfo.nTargetInputIndex     = 0;

// Generate a patch
returnCode = sceNgsPatchCreateRouting( &patchInfo, &patch );
```

Specify a voice output index for *`nSourceOutputIndex`*. Up to 4 voices can be output and a value of 0 to 3 can be specified for the index.

Specify the output index of a patch for *`nSourceOutputSubIndex`*. For example, you can use this index when you want to patch(connect) a single voice output with multiple voices.

Specify a voice input index for *`nTargetInputIndex`*.

When creating a patch for a voice that has 2 or more input mixers such as the side chain compressor, specify this index to indicate which voice output you want to connect with which input mixer.

Specify 0 when there is a single input mixer like the reverb buss voice.

Playing back the voice

You can now play back the voice you have set.

Please playback not only the voice for the generator, but also the voice of the data processor because it is required to run the voice.

```
returnCode = sceNgsVoicePlay( voicePlayer );
returnCode = sceNgsVoicePlay( s_voiceMaster );
```

Updating the NGS system

To start the NGS processing and update the parameters, you need to update the NGS system.

To do so, use `sceNgsSystemUpdate()` as described below. The parameters you set are not immediately reflected to the NGS system, but done when this function is called.

```
returnCode = sceNgsSystemUpdate( s_sysHandle );
```

Also note that a new PCM audio packet is generated following the call to this function.

The audio can be output by obtaining the output from the master buss and passing the data to the audio output library.

```
returnCode=sceNgsVoiceGetStateData(s_voiceMaster,
SCE_NGS_MASTER_BUSS_OUTPUT_MODULE, outputData,
sizeof(short) * SYS_GRANULARITY * 2 );
```

Note: Any "Locked" voices will be processed using the parameter values from their "pre-Locked" state. (`sceNgsSystemUpdate()` will only use new parameter values once a voices module is "Unlocked")

Terminating the NGS system

When all the processings of NGS are completed, terminate the NGS system as shown below.

```
// Shutdown NGS
returnCode = sceNgsSystemRelease( s_sysHandle );
```

5 Audio Playback with NGS

Streaming playback

As described in the chapter 4 "Basic Usage of NGS", the voice_template_1 voice and player module within atrac9_voice have a multiple buffer system and four buffers can be queued. These four buffers can be controlled and streaming playback can be performed. In addition, these buffers can be used for flexible playback beyond streaming.

Streaming playback, which is performed with the adpcm_stream sample provided in the following directory, is described using this sample.

- %SCE_PSP2_SDK_DIR%\target\samples\sample_code\audio_video\api_libngs\adpcm_stream

Although the basic procedure is the same as that described in the chapter 4 "Basic Usage of NGS", a detailed description of the player module setting is provided here. The buffer parameters of the player module can be used to specify the buffer address and size, the number of loops, and the next buffer. After the NGS performs playback from the buffer address at the size and number of loops specified here, buffer playback specified in the next buffer is performed. Specifying `SCE_NGS_PLAYER_NO_NEXT_BUFFER` to the next buffer parameter ends voice playback.

In the adpcm_stream sample, the following player module parameters are set. The initial values are set as follows; however, these parameters can be rewritten by runtime.

```
// Get player parameters
returnCode = sceNgsVoiceLockParams( hVoice,
                            nModuleId,
                            SCE_NGS_PLAYER_PARAMS_STRUCT_ID,
                            &bufferInfo );
if ( returnCode != SCE_NGS_OK )     {
printf( "initPlayer: sceNgsVoiceLockParams() failed: 0x%08X\n", returnCode );
return returnCode;
}

// Set player parameters
memset( bufferInfo.data, 0, bufferInfo.size);
pPcmParams = (SceNgsPlayerParams *)bufferInfo.data;
pPcmParams->desc.id     = SCE_NGS_PLAYER_PARAMS_STRUCT_ID;
pPcmParams->desc.size   = sizeof(SceNgsPlayerParams);

pPcmParams->fPlaybackFrequency = (SceFloat32)pSound->nSampleRate;
pPcmParams->fPlaybackScalar    = 1.0f;
pPcmParams->nLeadInSamples     = 0;
pPcmParams->nChannels         = pSound->nNumChannels;
if ( pSound->nNumChannels == 1 ) {
pPcmParams->nChannelMap[0] = SCE_NGS_PLAYER_LEFT_CHANNEL;
pPcmParams->nChannelMap[1] = SCE_NGS_PLAYER_LEFT_CHANNEL;
} else {
pPcmParams->nChannelMap[0] = SCE_NGS_PLAYER_LEFT_CHANNEL;
pPcmParams->nChannelMap[1] = SCE_NGS_PLAYER_RIGHT_CHANNEL;
}
pPcmParams->nType           = pSound->nType;
pPcmParams->nRespectLoopFlag = 1;

pPcmParams->buffs[0].pBuffer    = s_ptrStreamBuffer[0];
pPcmParams->buffs[0].nNumBytes  = STREAM_BUFFER_SIZE;
pPcmParams->buffs[0].nLoopCount = 0;
pPcmParams->buffs[0].nNextBuff  = 1;
pPcmParams->buffs[1].pBuffer    = s_ptrStreamBuffer[1];
```

```
pPcmParams->buffs[1].nNumBytes  = STREAM_BUFFER_SIZE;
pPcmParams->buffs[1].nLoopCount = 0;
pPcmParams->buffs[1].nNextBuff  = 0;

// Update player parameters
returnCode = sceNgsVoiceUnlockParams( hVoice, nModuleId );
if ( returnCode != SCE_NGS_OK ) {
printf( "initPlayer: sceNgsVoiceUnlockParams() failed: 0x%08X\n", returnCode );
if ( returnCode == SCE_NGS_ERROR_PARAM_OUT_OF_RANGE ) {
        printParamError( hVoice, nModuleId );
}
return returnCode;
}
```

In addition, the end of the buffer can be detected by setting a callback as shown below.

```
returnCode = sceNgsVoiceSetModuleCallback( s_voicePlayer,
                                   SCE_NGS_VOICE_T1_PCM_PLAYER,
                                   playerCallback, NULL );
if ( returnCode != SCE_OK ) {
return returnCode;
}
```

Streaming playback can be performed by setting the callback set here, as a trigger and rewriting the buffer and changing the parameters. In the adpcm_stream sample, the buffer is rewritten as follows within the callback.

Although only buffer rewriting is performed in the adpcm_stream sample, changes can be made here, including specifying the next buffer and setting the number of loops, thereby enabling flexible playback.

```
void playerCallback( const SceNgsCallbackInfo *pCallbackInfo )
{
        int              returnCode;
        SceNgsBufferInfo   buffInfo;
        SceNgsPlayerParams *pPcmParams;
        unsigned int      uBytesToRead;
        static int       s_nNextStreamBuffer = 2;

        // Determine how many bytes to copy
        if ( s_sound.nNumBytes - s_nBytesRead < STREAM_BUFFER_SIZE ) {
             uBytesToRead = s_sound.nNumBytes - s_nBytesRead;
        } else {
             uBytesToRead = STREAM_BUFFER_SIZE;
        }

        // Get player parameters
        returnCode = sceNgsVoiceLockParams( s_voicePlayer,
                                   SCE_NGS_VOICE_T1_PCM_PLAYER,
                                   SCE_NGS_PLAYER_PARAMS_STRUCT_ID,
                                   &buffInfo );
        if ( returnCode != SCE_NGS_OK ) {
             printf( "playerCallback: sceNgsVoiceLockParams() failed:
0x%08X\n", returnCode );
             return;
        }
        pPcmParams = (SceNgsPlayerParams *)buffInfo.data;

        // Copy data
        pPcmParams->buffs[s_nNextBuffer].pBuffer =
s_ptrStreamBuffer[s_nNextStreamBuffer];
        s_nNextStreamBuffer = (s_nNextStreamBuffer + 1) % NUM_STREAM_BUFFERS;
        memcpy( pPcmParams->buffs[s_nNextBuffer].pBuffer, (char
*)s_sound.pData + s_nBytesRead, uBytesToRead );
```

```
            pPcmParams->buffs[s_nNextBuffer].nNumBytes = uBytesToRead;

            s_nNextBuffer = (s_nNextBuffer + 1) % 2;
            s_nBytesRead += uBytesToRead;
            if ( s_nBytesRead == s_sound.nNumBytes ) {
                  s_nBytesRead = 0;
            }

            // Update player parameters
            returnCode = sceNgsVoiceUnlockParams( s_voicePlayer,
   SCE_NGS_VOICE_T1_PCM_PLAYER );
            if ( returnCode != SCE_NGS_OK ) {
                  printf( "playerCallback: sceNgsVoiceUnlockParams() failed:
   0x%08X\n", returnCode );
                  if ( returnCode == SCE_NGS_ERROR_PARAM_OUT_OF_RANGE ) {
                         printParamError( s_voicePlayer,
   SCE_NGS_VOICE_T1_PCM_PLAYER );
                  }
                  return;
            }
```