NintendoWare for CTR

Sound Programmer's Guide

2011/01/12

Ver. 1.6.1

PROVISIONAL TRANSLATION

The content of this document is highly confidential and should be handled accordingly.

Table of Contents

1 Introduction 8
1.1 Supported Platforms 8
2 The Sound Program Development Environment 9
2.1 The nw::snd Library 9
2.2 Directory Structure 9
2.3 Development Environment Used with the nw::snd Library 10
2.3.1 Configure the NintendoWare Build Environment 10
2.3.2 Configuring the Library File Links 10
2.3.3 Configuring the Include Path 10
2.3.4 Placing the Sound Data 10
2.3.5 Reference the Sound ID File 10
3 Quick Start 11
3.1 Building and Running the Sample Demo 11
3.1.1 Location of the Sample Demos 11
3.1.2 Build the Sample Demo 11
3.1.3 Start the Sample Demo 11
3.1.4 SoundMaker Project Data 12
3.1.5 Playing and Stopping Sounds 12
3.2 Source Code 13
3.2.1 Referencing Header Files 17
3.2.2 Namespace 17
3.3 The Initialization Process 17
3.3.1 Initialization of the Sound System 17
3.3.2 Initialization of the Sound Archive 18
3.3.3 Initialization of the Sound Data Manager 18
3.3.4 Initialization of the Sound Archive Player 19
3.3.5 Construction of the Sound Heap 20
3.4 Loading Sound Data 20
3.5 Frame Processing 21
3.6 Starting and Stopping Sound Playback 21
3.6.1 Sound Playback 21
3.6.2 Sound Handles 21
3.6.3 Stopping Sound Playback 22
3.7 Playback Using the HoldSound Function 22
3.7.1 Using the HoldSound Function 22
3.7.2 Resume Playback After Losing Priority 23
3.7.3 Priority Processing with the HoldSound Function 23

3.8 Playback Using the PrepareSound Function 23
3.8.1 Using the PrepareSound Function 23
3.9 Playback Using a String 24
3.10 Types of Sounds 25
4 Memory Management 26
4.1 How to Use the Sound Heap 26
4.1.1 Clear All 26
4.1.2 Restore Prior State 26
4.2 Sound Heap Application 27
4.2.1 Feature to Automatically Stop Invalid Sound 27
4.2.2 Multiple Sound Heaps 27
4.2.3 SoundMemoryAllocatable Class 28
4.3 Player Heap 28
4.3.1 The Player Heap 28
4.3.2 Using the Player Heap 28
4.3.3 Memory for the Player Heap 29
4.3.4 Cautions for Playback with Different Players 29
5 Sound Actors 30
5.1 Sound Actors 30
5.2 Executing the Sample Demo 30
5.2.1 Starting the Sample Demo 30
5.2.2 Controls 30
5.3 The `SoundActor` Program 31
5.3.1 Initializing the Instances 31
5.3.2 Sound Playback Using Sound Actors 32
5.3.3 Setting the Maximum Number of Sounds to Be Played Simultaneously 32
5.3.4 Parameter Updates 32
5.3.5 Deallocating Sound Actors 32
5.4 Actor Players 32
5.4.1 Maximum Number of Simultaneous Sounds Played and the Actor Player 32
5.4.2 Setting the Number of the Actor Player to Be Played 33
5.5 Dynamically Changing the Sound ID 33
5.5.1 The `SoundActor::SetupSound` Function 34
5.5.2 Changing the Sound ID 34
6 3D Sound 36
6.1 3D Sound Structure 36
6.1.1 The 3D Sound Manager (Sound3DManager) 36
6.1.2 The 3D Sound Actor (Sound3DActor) 36
6.1.3 The 3D Sound Listener (Sound3DListener) 36

6.2 Sample Demo Execution 36
6.2.1 Start the Sample Demo 36
6.2.2 Controls 36
6.3 3D Sound Program 37
6.3.1 Creating Instances 38
6.3.2 Initializing the 3D Sound Manager 38
6.3.3 Initializing the Listener 39
6.3.4 Initialize the Actor 41
6.3.5 3D Sound Playback 42
6.3.6 Updating the Actor's Coordinates 42
6.3.7 Lifespans of Actors and Sounds 42
6.4 Doppler Effect 43
6.4.1 Doppler Effect Parameters 43
6.4.2 Changing the Tone 44
6.4.3 Explicitly Specifying Speed 44
6.4.4 Formula for Pitch Change 44
6.5 Multi-Listeners 45
6.5.1 Differences Compared to a Single Listener 45
6.6 3D Sound Customization 45
6.6.1 3D Sound Engines (Sound3DEngine) 45
6.6.2 Creating a 3D Sound Engine 45
6.6.3 Implementing 3D Sound Calculation Processing 46
6.6.4 Sound3DCalculator 47
7 Notes About Using the nn::snd Library 49
7.1 nn::snd::AllocVoice 49
7.1.1 Precautions When Threads Are Running in the System Core 49
7.1.2 Synchronous Processing 50
7.2 nn::snd::WaitForDspSync, nn::snd::SendParameterToDsp 50
7.3 Matching the Timing of Parameter Settings 51
7.3.1 Use nw::snd::SoundSystem::SetSoundFrameUserCallback 51
7.3.2 Call nn::snd::WaitForDspSync or nn::snd::SendParameterToDsp from the application 51
7.4 nn::snd::SetMasterVolume 51
8 Operation in the System Core 52
8.1 What is the System Core? 52
8.2 How to Operate in the System Core 52
8.3 Processes That Can Run in the System Core 53
8.4 Limitations when Operating in System Core 53
8.5 Supplemental Information 53
9 Processing During Sleep 55

9.1 Restrictions During Sleep 55
9.2 nw::snd::SoundSystem::EnterSleep, nw::snd::SoundSystem::LeaveSleep 55
10 Revision History 57

Code

Code 2-1 Library File Linking Configuration 10
Code 3-1 Building the Simple Demo 11
Code 3-3 `simple` Demo SimpleApp.cpp 14
Code 3-4 Reference Header Files 17
Code 3-5 Initializing the Sound System 17
Code 3-6 Initializing the Sound Archive 18
Code 3-7 Initializing the Sound Data Manager 19
Code 3-8 Initializing the Sound Archive Player 19
Code 3-9 Constructing the Sound Heap 20
Code 3-10 Loading Sound Data 20
Code 3-11 Frame Processing 21
Code 3-12 Sound Playback 21
Code 3-13 Stopping Sound Playback 22
Code 3-14 Playback Using a Sound ID 24
Code 3-15 Playback Using a String 24
Code 3-16 Loading Label String Data 24
Code 3-17 Using Sequence Sound Handles 25
Code 5-1 `SoundActor` Instance Initialization 31
Code 5-2 Sound Playback Using Sound Actors 32
Code 5-3 Setting the Maximum Number of Sounds to Be Played Simultaneously 32
Code 5-4 Parameters Updates 32
Code 5-5 The `SoundActor::SetupSound` Function 34
Code 5-6 Changing the Sound ID 34
Code 6-1 Instance Initialization 38
Code 6-2 3D Sound Manager Initialization 38
Code 6-3 Initializing the Listener 39
Code 6-4 Calculating the Listener Matrix 40
Code 6-5 Initialize the Actor 42
Code 6-6 3D Sound Playback 42
Code 6-7 Updating Actor Coordinates 42
Code 6-8 Setting the Speed of Sound 43
Code 6-9 3D Sound Engine Inheritance 45
Code 6-10 3D Sound Engine Registration 45
Code 6-11 Overriding the UpdateAmbientParam Function 46
Code 6-12 The `SoundAmbientParam` Structure 46

Code 6-13 Sound3DCalculator Class 47
Code 7-1 nw::snd Limiting Number of Voices Used By Library 49
Code 8-1 Operation in the System Core 52
Code 9-1 Functions Called During Sleep 55
Code 9-2 Sleep Process Sample Code 55

Tables

Table 3-1 Button Assignments for the `simple` Demo 13
Table 5-1 `soundActor` Demo Button Assignment 30
Table 6-1 `sound3d` Demo Button Assignment 37
Table 8-1 Relation between cores where sound thread is operating and effect is operating 53

Figures

Figure 2-1 Directory Structure 9
Figure 3-1 `simple` Demo Screens 12
Figure 3-2 Sound and Sound Handle Are Tied Together When Sound Playback Succeeds 22
Figure 4-1 The LoadState Function Can Be Used to Restore a Prior State 27
Figure 5-1 `soundActor` Demo Screens 31
Figure 5-1 Actor Player 33
Figure 6-1 Reducing the Priority Value 39
Figure 6-3 Attenuation Unit Distance 41
Figure 6-4 Doppler Effect 43
Figure 6-5 Automatic Calculation of Speed 44

1 Introduction

The nw::snd library of NintendoWare for CTR (NintendoWare) is used for programming sounds for game software for CTR. Use of the Library enables sound data created with SoundMaker and other NintendoWare sound tools to be played on the CTR system.

This document explains how to use the nw::snd Library to assemble sound programs. It first describes how to build the sound program development environment,and then shows sample demos to demonstrate specific instructions for using sound programs.

Before reading this document, make sure you have read NintendoWare_ProgrammersGuide.pdf and understand the basics of the NintendoWare runtime libraries.

For more information about the individual classes and functions, see the *Function Reference Manual*.

1.1 Supported Platforms

The nw::snd library targets the CTR system and can be used on the CTR-TEG2 board.

You can also use the library for development on a Windows PC. However, applications that run on a Windows PC might not run as is on the CTR system.

This document describes the CTR-TEG2 environment settings and build procedure. See the *Programmer's Guide* for details on the Windows PC environment settings and build procedure.

2 The Sound Program Development Environment

This chapter explains how to build the sound program development environment.

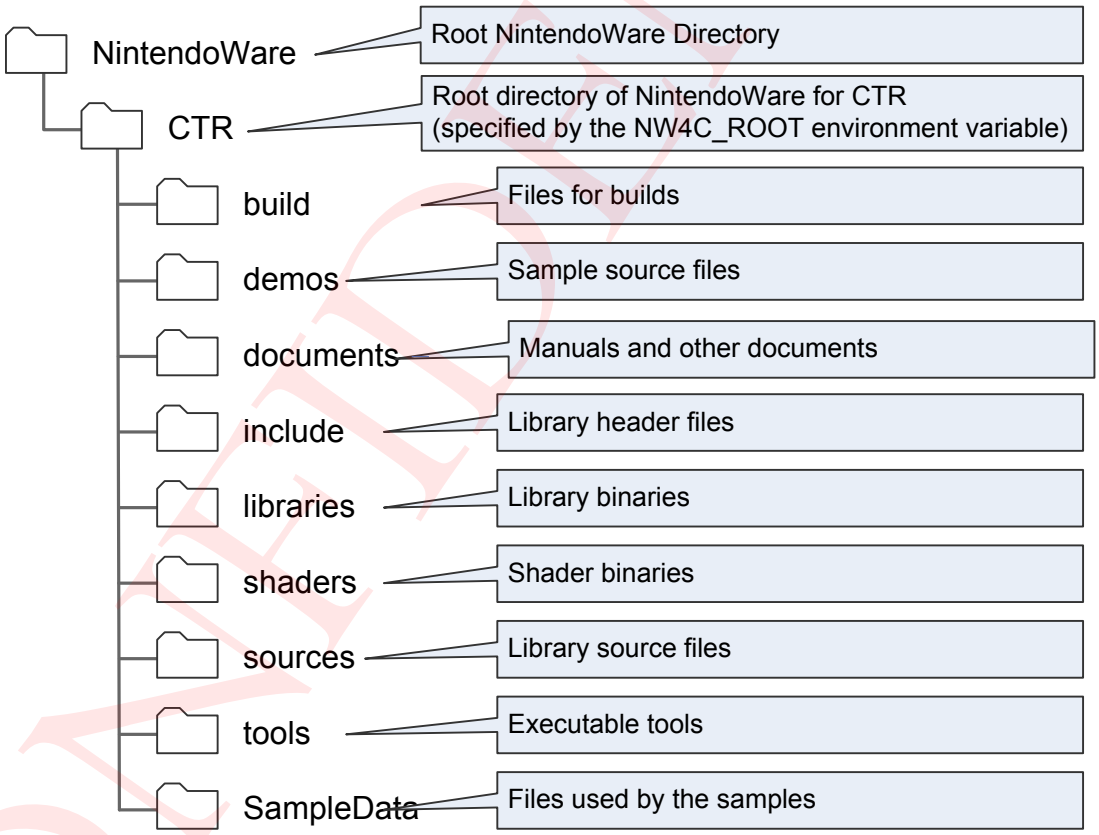
2.1 The nw::snd Library

The nw::snd Library is used to assemble sound programs for the Revolution. The library is located in the NintendoWare package.

2.2 Directory Structure

The following figure shows the directory structure of the nw::snd Library. $NW4C_ROOT denotes the NintendoWare/CTR installation directory.

Figure 2-1 Directory Structure



2.3 Development Environment Used with the nw::snd Library

This section describes how to prepare the development environment to develop sounds using the nw::snd Library.

2.3.1 Configure the NintendoWare Build Environment

You must first construct the NintendoWare build environment. See the *Programmer's Guide* for details.

2.3.2 Configuring the Library File Links

To use the nw::snd Library, you must link the nw::snd Library and any library files that depend on it. Use a linking configuration similar to the following in your OMakefile.

Code 2-1 Library File Linking Configuration

```
LIBS += libnn_dsp libnn_dspsnd

NW_LIBRARIES[] =
    libnw_snd
    libnw_io
    libnw_os
    libnw_ut
```

As of version NW4C-0.4.1, the nw::snd Library depends on the CTR SDK libraries dsp and dspsnd, and on the NW4C libraries nw::io, nw::os, and nw::ut.

2.3.3 Configuring the Include Path

Because the include path is a shared path in NintendoWare, it is not necessary to set a special include path to use the nw::snd Library.

2.3.4 Placing the Sound Data

The sound data created by the sound designer is passed to the programmer as a Sound Archive (*.bcsar) and a folder containing sets of stream data (*.bcstm). The Sound Archive is a single file that groups sets of sound data, but not stream data.

To use this sound data, the programmer should place it in the ROM file system. To do so, the necessary files should be located within the directory specified in the OMakefile by ROMFS_ROOT.

2.3.5 Reference the Sound ID File

The Sound ID file (*.csid) contains definitions of labels for using the Sound Archive. This file is created when the Sound Archive is created. The programmer includes this by referencing it in the source file.

3 Quick Start

This chapter uses a sample demo to demonstrate how to build a simple sound program.

First, you will build and execute the sample demo, and then you will look at the sample's source code while reviewing how to assemble a sound program.

3.1 Building and Running the Sample Demo

3.1.1 Location of the Sample Demos

The sample demos are located in the `$NW4C_ROOT/demos/snd` directory. For this demo, the file called "`simple`" is used.

3.1.2 Build the Sample Demo

To build the sample demo, move to the directory where `simple` resides and run `omake`.

Code 3-1 Building the Simple Demo

```
cd $NW4C_ROOT/demos/snd/simple
$ omake
```

If the build is successful, a `simple.cci` file will be created in `simple/images/CTR-TEG2.Process.MPCore.fast/Development/` directory (the directory name may change depending on your build options).

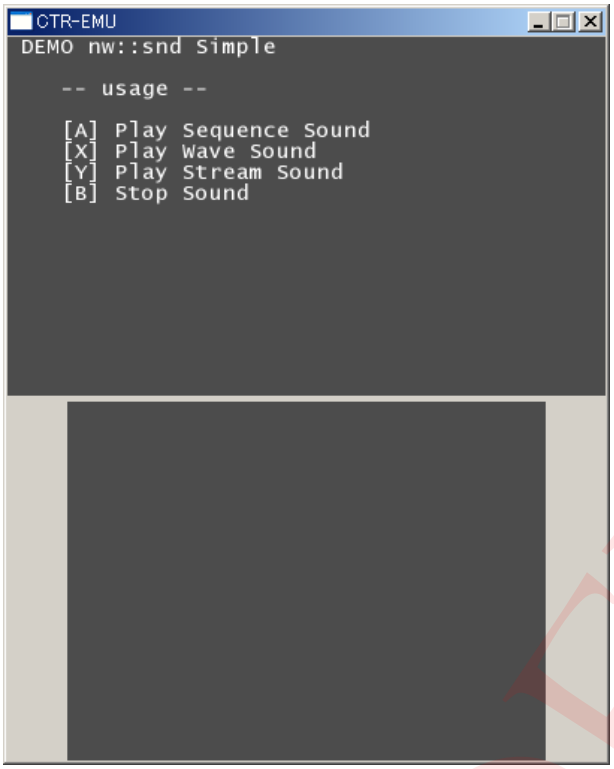
3.1.3 Start the Sample Demo

To execute the sample demo, load the generated `simple.cci` file into the `PARTNER-CTR/S` debugger. From the `PARTNER-CTR/S` debugger, click **File** → **Load**,and select the `simple.cci` file.

Note: It may take some time to load the file.

Next, click **Run** → **Run Program (G)** to run the sample program. If it launches properly, the LCD screens on the CTR-TEG2 board should look as follows.

Figure 3-1 `simple` Demo Screens



Note: The screenshot above is from a demo running on a Windows PC.It may differ from the actual display shown on a CTR-TEG2 board.

3.1.4 SoundMaker Project Data

The data used by the `simple` demo is converted data stored in the `$NW4C_ROOT/demos/demolib/data/` directory. The SoundMaker project data is stored in the `$NW4C_ROOT/tools/SoundMaker/samples/simple/` directory before conversion, so reference it there if necessary.

3.1.5 Playing and Stopping Sounds

To start and stop the playback of sounds, use the PC keyboard (for the PC demo version) or the buttons on the CTR-TEG2 (for the CTR-TEG2 demo version). Table 3-1 lists the operations assigned to each button.

Table 3-1 Button Assignments for the `simple` Demo

Button	Operation
A Button	Play sequence sound
Y Button	Play stream sound
X Button	Play wave sound
B Button	Stop sound

3.2 Source Code

This section explains the sound program by showing the contents of the source code. The code below is for the `SimpleApp.h` and `SimpleApp.cpp` files located in the `$NW4C_ROOT/demos/snd/simple/sources/` directory.

Code 3-2 `simple` Demo SimpleApp.h

```
#include "demolib.h"
#include <nw/snd.h>
class SimpleApp : public nw::snd::demolib::AppBase
{
protected:
   virtual void OnInitialize();
   virtual void OnFinalize();
   virtual void OnUpdatePad( nw::demo::Pad& );
   virtual void OnUpdate();

private:
   void InitializeSoundSystem();

   nw::snd::RomSoundArchive   m_Archive;
   nw::snd::SoundArchivePlayer m_ArchivePlayer;
   nw::snd::SoundDataManager   m_DataManager;
   nw::snd::SoundHeap         m_Heap;
   nw::snd::SoundHandle       m_Handle;

   void* m_pMemoryForSoundSystem;
   void* m_pMemoryForInfoBlock;
   void* m_pMemoryForSoundDataManager;
   void* m_pMemoryForSoundArchivePlayer;
   void* m_pMemoryForSoundHeap;
   void* m_pMemoryForStreamBuffer;
};
```

Code 3-3 `simple` Demo SimpleApp.cpp

```
#include "precompiled.h"
#include "SimpleApp.h"
#include "simple.csid"

namespace
{
const s32 SOUND_THREAD_PRIORITY = 4;
const s32 LOAD_THREAD_PRIORITY = 3;
const s32 SOUND_HEAP_SIZE = 1 * 1024 * 1024;
const char SOUND_ARC_PATH[] = NW_SND_DEMO_PATH_PREFIX "simple.bcsar";
}

void SimpleApp::OnInitialize()
{
   InitializeSoundSystem();

   // Load sound data
   if ( ! m_DataManager.LoadData( SEQ_MARIOKART, &m_Heap ) )
   {
       NW_ASSERTMSG( false, "LoadData(SEQ_MARIOKART) failed." );
   }
   if ( ! m_DataManager.LoadData( SE_YOSHI, &m_Heap ) )
   {
       NW_ASSERTMSG( false, "LoadData(SE_YOSHI) failed." );
   }
}

void SimpleApp::InitializeSoundSystem()
{
   // Initialize sound system
   {
       nw::snd::SoundSystem::SoundSystemParam param;
       size_t workMemSize = nw::snd::SoundSystem::GetRequiredMemSize( param );
       m_pMemoryForSoundSystem = MemAlloc( workMemSize );

       nw::snd::SoundSystem::Initialize(
              param,
              reinterpret_cast<uptr>( m_pMemoryForSoundSystem ),
              workMemSize );
   }
```

```
// Initialize sound archive
if ( ! m_Archive.Open( SOUND_ARC_PATH ) )
{
    NW_ASSERTMSG( 0, "cannot open bcsar(%s)¥n", SOUND_ARC_PATH );
}

// INFO block download
{
    size_t infoBlockSize = m_Archive.GetHeaderSize();
    m_pMemoryForInfoBlock = MemAlloc( infoBlockSize );
    if ( ! m_Archive.LoadHeader( m_pMemoryForInfoBlock, infoBlockSize ) )
    {
        NW_ASSERTMSG( 0, "cannot load infoBlock(%s)", SOUND_ARC_PATH );
    }
}

// Initialize sound data manager
{
    size_t setupSize = m_DataManager.GetRequiredMemSize( &m_Archive );
    m_pMemoryForSoundDataManager = MemAlloc( setupSize );
    m_DataManager.Initialize(
           &m_Archive, m_pMemoryForSoundDataManager, setupSize );
}

// Initialize sound archive player
{
    size_t setupSize = m_ArchivePlayer.GetRequiredMemSize( &m_Archive );
    m_pMemoryForSoundArchivePlayer = MemAlloc( setupSize );
    size_t setupStrmBufferSize =
       m_ArchivePlayer.GetRequiredStreamBufferSize( &m_Archive );
    m_pMemoryForStreamBuffer = MemAlloc( setupStrmBufferSize, 32 );
    bool result = m_ArchivePlayer.Initialize(
           &m_Archive,
           &m_DataManager,
           m_pMemoryForSoundArchivePlayer, setupSize,
           m_pMemoryForStreamBuffer, setupStrmBufferSize );
    NW_ASSERT( result );
}

// Allocate sound heap
{
    m_pMemoryForSoundHeap = MemAlloc( SOUND_HEAP_SIZE );
```

```
        bool result = m_Heap.Create( m_pMemoryForSoundHeap, SOUND_HEAP_SIZE );
        NW_ASSERT( result );
    }
}

void SimpleApp::OnUpdatePad( nw::demo::Pad& pad )
{
    if ( pad.IsButtonDown( nw::demo::Pad::BUTTON_A ) )
    {
        m_Handle.Stop( 0 );
        bool result = m_ArchivePlayer.StartSound(
              &m_Handle, SEQ_MARIOKART ).IsSuccess();
        NN_LOG("[SEQ] SEQ_MARIOKART ... (%d)¥n", result);
    }

    if ( pad.IsButtonDown( nw::demo::Pad::BUTTON_X ) )
    {
        m_Handle.Stop( 0 );
        bool result = m_ArchivePlayer.StartSound(
              &m_Handle, SE_YOSHI ).IsSuccess();
        NN_LOG("[WSD] SE_YOSHI ... (%d)¥n", result);
    }

    if ( pad.IsButtonDown( nw::demo::Pad::BUTTON_Y ) )
    {
        m_Handle.Stop( 0 );
        bool result = m_ArchivePlayer.StartSound(
              &m_Handle, STRM_MARIOKART ).IsSuccess();
        NN_LOG("[STRM] STRM_MARIOKART ... (%d)¥n", result );
    }

    if ( pad.IsButtonDown( nw::demo::Pad::BUTTON_B ) )
    {
        m_Handle.Stop( 3 );
    }
}

void SimpleApp::OnUpdate()
{
    m_ArchivePlayer.Update();
```

3.2.1 Referencing Header Files

The `SimpleApp.h` and `SimpleApp.cpp` files reference the following header files.

Code 3-4 Reference Header Files

```
#include <nw/snd.h>
#include "simple.csid"
```

The `nw/snd.h` file is a header file for the `nw::snd` Library. This header file must be referenced in order to use the snd Library.

The `simple.csid` file is a Sound ID file. By using the labels defined in this file, you can use specific sets of sound data in the Sound Archive.

3.2.2 Namespace

The `nw::snd` Library is defined by the namespace `nw::snd`.

3.3 The Initialization Process

The basic initialization process involves the following steps.

1. Initialization of the Sound System
2. Initialization of the Sound Archive
3. Initialization of the Sound Data Manager
4. Initialization of the Sound Archive Player
5. Construction of the Sound Heap

Each of these processes is explained in order.

3.3.1 Initialization of the Sound System

To initialize the Sound System, call the following functions:

Code 3-5 Initializing the Sound System

```
// Initialize sound system
{
    nw::snd::SoundSystem::SoundSystemParam param;
    size_t workMemSize = nw::snd::SoundSystem::GetRequiredMemSize( param );
    m_pMemoryForSoundSystem = MemAlloc( workMemSize );

    nw::snd::SoundSystem::Initialize(
            param,
            reinterpret_cast<uptr>( m_pMemoryForSoundSystem ),
            workMemSize );
}
```

The nw::snd::SoundSystem::Initialize function initializes the Sound System. The arguments specify the priority of the sound thread and the sound data load thread, and also pass an nw::snd::SoundSystem::SoundSystemParam structure holding the stack sizes of each of the threads. The sound thread handles sound playback. To prevent delays in playback, the sound thread must be set to a high priority.

The sound data load thread handles stream data loading. If there are delays in the loading of stream data there will be interruptions in sound, so the sound data load thread must also be set to a high priority. To prevent sound breaks, we recommend setting the sound data load thread to a higher priority than the sound thread.

3.3.2 Initialization of the Sound Archive

Next the Sound Archive is initialized, as shown below.

Code 3-6 Initializing the Sound Archive

```
// Initialize sound archive
if ( ! m_Archive.Open( SOUND_ARC_PATH ) )
{
    NW_ASSERTMSG( 0, "cannot open bcsar(%s)¥n", SOUND_ARC_PATH );
}

// Load INFO block
{
    size_t infoBlockSize = m_Archive.GetHeaderSize();
    m_pMemoryForInfoBlock = MemAlloc( infoBlockSize );
    if ( ! m_Archive.LoadHeader( m_pMemoryForInfoBlock, infoBlockSize ) )
    {
        NW_ASSERTMSG( 0, "cannot load infoBlock(%s)", SOUND_ARC_PATH );
    }
}
```

The nw::snd::RomSoundArchive class instance m_Archive is created ahead of time. Open the Sound Archive using the nw::snd::RomSoundArchive::Open function. The argument specifies the path to the ROM filesystem.

Next, call the nw::snd::RomSoundArchive::LoadHeader function to load the minimum essential amount of information. The size of memory required for loading this information is passed as one of the function's arguments. Use the nw::snd::RomSoundArchive::GetHeaderSize function to get the required size of this region.

3.3.3 Initialization of the Sound Data Manager

The next step is to initialize the Sound Data Manager. This is the class for loading and managing the data stored in a sound archive.

Code 3-7 Initializing the Sound Data Manager

```
// Initialize the sound data manager
{
    size_t setupSize = m_DataManager.GetRequiredMemSize( &m_Archive );
    m_pMemoryForSoundDataManager = MemAlloc( setupSize );
    m_DataManager.Initialize(
          &m_Archive, m_pMemoryForSoundDataManager, setupSize );
}
```

The `nw::snd::SoundDataManager` class instance `m_DataManager` is created ahead of time. Use the `nw::snd::SoundDataManager::Initialize` function to initialize the Sound Data Manager. The function arguments take `&m_Archive`, a pointer to the sound archive, and the size of the memory required to initialize the Sound Data Manager. Use the `nw::snd::SoundDataManager::GetRequiredMemSize` function to get the required size of this region.

3.3.4 Initialization of the Sound Archive Player

The next step is to initialize the Sound Archive Player. This is the class for using the Sound Archive to play sounds.

Code 3-8 Initializing the Sound Archive Player

```
// Initialize Sound Archive Player
{
    size_t setupSize = m_ArchivePlayer.GetRequiredMemSize( &m_Archive );
    m_pMemoryForSoundArchivePlayer = MemAlloc( setupSize );
    size_t setupStrmBufferSize =
       m_ArchivePlayer.GetRequiredStreamBufferSize( &m_Archive );
    m_pMemoryForStreamBuffer = MemAlloc( setupStrmBufferSize, 32 );
    bool result = m_ArchivePlayer.Initialize(
          &m_Archive,
          &m_DataManager,
          m_pMemoryForSoundArchivePlayer, setupSize,
          m_pMemoryForStreamBuffer, setupStrmBufferSize );
    NW_ASSERT( result );
}
```

The `nw::snd::SoundArchivePlayer` class instance `m_ArchivePlayer` is created ahead of time.

The Sound Archive Player is set up by the `nw::snd::SoundArchivePlayer::Initialize` function. The function arguments are: the `&m_Archive` pointer to the Sound Archive, the `&m_DataManager` pointer to the Sound Data Manager, and two memory regions required for setup. The first memory region is used as a work area and contains instances required by the library, and the second is used as a stream playback buffer.

The required memory size can be obtained using nw::snd::SoundArchivePlayer::GetRequiredMemSize and nw::snd::SoundArchivePlayer::GetRequiredStreamBufferSize.

3.3.5 Construction of the Sound Heap

The final process is to construct the sound heap. This is the class for managing the memory regions for loading sound data.

Code 3-9 Constructing the Sound Heap

```
// Construct the sound heap
{
    m_pMemoryForSoundHeap = MemAlloc( SOUND_HEAP_SIZE );
    bool result = m_Heap.Create( m_pMemoryForSoundHeap, SOUND_HEAP_SIZE );
    NW_ASSERT( result );
}
```

The nw::snd::SoundHeap class instance m_Heap is created ahead of time.

The sound heap is built by the nw::snd::SoundHeap::Create function. The function takes the memory region assigned to the sound heap as an argument.

3.4 Loading Sound Data

Normally, before a sound can be played, the required sound data must be loaded. The sound data is loaded in groups, and can also be loaded as sequence data, wave sound data, bank data, and waveform archive data. These groups are created by the sound designer when the sound data is created.

Code 3-10 Loading Sound Data

```
// Load sound data
if ( ! m_DataManager.LoadData( SEQ_MARIOKART, &m_Heap ) )
{
    NW_ASSERTMSG( false, "LoadData(SEQ_MARIOKART) failed." );
}
if ( ! m_DataManager.LoadData( SE_YOSHI, &m_Heap ) )
{
    NW_ASSERTMSG( false, "LoadData(SE_YOSHI) failed." );
}
```

The nw::snd::SoundDataManager::LoadData function loads the sound data. The arguments take the sound data to load or the group's label, and the sound heap for storing the loaded data.

The group labels SEQ_MARIOKART and SE_YOSHI are defined by the Sound ID file simple.csid. Confirm with the sound designer which data should be loaded at which times.

The simple demo loads all data for the sequence sound SEQ_MARIOKART (sequence data, bank data, waveform archive data) and all data for the wave sound SE_YOSHI (wave sound data, waveform archive data).

Note: The nw::snd::SoundDataManager::LoadData function performs synchronous loading. No asynchronous version of this function is provided. To perform asynchronous loading, call this function from another thread. Be sure to refer to the reference manual for precautions when calling the nw::snd::SoundDataManager::LoadData function from another thread.

3.5 Frame Processing

The Sound Archive Player must be updated in every frame.

Code 3-11 Frame Processing

```
m_ArchivePlayer.Update();
```

This update process is normally called from the main loop. It does not need to be called in every video frame.

3.6 Starting and Stopping Sound Playback

3.6.1 Sound Playback

Sounds are played using code like the following.

Code 3-12 Sound Playback

```
bool result = m_ArchivePlayer.StartSound(
      &m_Handle, SEQ_MARIOKART ).IsSuccess();
```

The nw::snd::SoundArchivePlayer::StartSound function plays sounds. The arguments take the sound handle and the label for the sound being played.

The nw::snd::SoundHandle class instance pointer &m_Handle is passed as the sound handle (explained in section 3.6.2 Sound Handles).

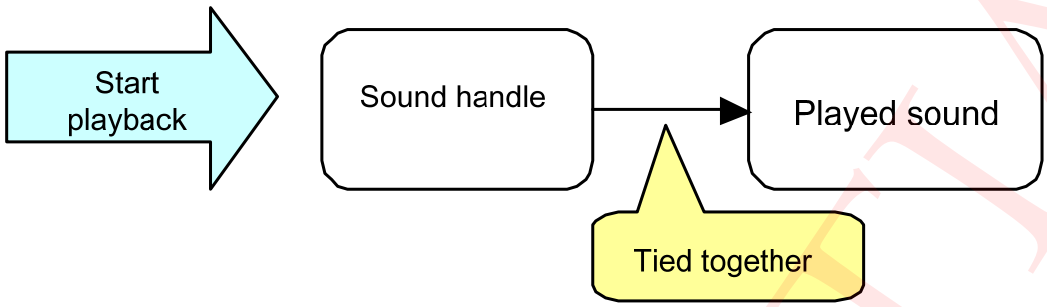
The label for the sound being played is defined in the Sound ID file simple.csid. You need to confirm with the sound designer which label is for which sound.

3.6.2 Sound Handles

3.6.2.1 What Is a Sound Handle?

A sound handle is an object for stopping sounds that are playing, changing the volume, and other functions. Each sound handle can control one sound. When the sound starts playing successfully, the sound and a sound handle become tied together. Until that tie is cut, the sound handle can be used to control that sound.

Figure 3-2 Sound and Sound Handle Are Tied Together When Sound Playback Succeeds



This means that the programmer does not need to be concerned with whether a sound is still playing or not. You can carry out the same process on a sound whether it is playing or already stopped without mistakenly performing the operation on a different sound.

3.6.2.2 Tips for Creation of Sound Handles

For sounds like one-time sound effects that play without stopping or being manipulated, you only need to prepare one sound handle, using it repeatedly to play each sound in turn. Also, you can use it immediately after a sound to change the volume and other parameters before playing the next sound.

For persistent effects like background music and engine noise, you'll at least need to stop the sound at some point, so you will need a sound handle for each individual sound.

3.6.3 Stopping Sound Playback

A sound handle is used to stop its associated sound.

Code 3-13 Stopping Sound Playback

```
m_Handle.Stop( 3 );
```

Use the `nw::snd::SoundHandle::Stop` function to stop sounds. The argument takes the number of fadeout frames. The volume is gradually decreased for the duration of the specified number of fadeout frames and then playback is stopped.

As mentioned above, it is okay if the sound stops before the function is called. If this occurs, the function returns without doing anything.

3.7 Playback Using the HoldSound Function

Normally, the `StartSound` and `Stop` functions are used to play back and stop sound. However, it is sometimes troublesome to call the `Stop` function at the appropriate time when implementing the program. In these cases, the `HoldSound` function can be used in place of the `StartSound` function.

3.7.1 Using the HoldSound Function

When using the `HoldSound` function, it must be called for each frame while the sound is playing. As this happens, the playback start process is performed the first time only, and no processing is done

the second or subsequent times. When the continued callback is stopped, the sound playback is automatically stopped in the `snd::SoundArchivePlayer::Update` function.

The holdSound sample using the `HoldSound` function is also available, so please refer to it as well.

3.7.2 Resume Playback After Losing Priority

When `StartSound` and `Stop` are used for playback, if sound playback is forcibly stopped due to loss of priority, playback will not resume unless `StartSound` is explicitly called.

When using `HoldSound`, even if priority is lost, an attempt to resume playback will occur in the next frame because the function is called in each frame. For this reason, playback automatically resumes when the playback of a higher priority sound completes.

However, because playback resumes from the beginning rather than from the middle, be aware that some sound data may sound unnatural.

3.7.3 Priority Processing with the HoldSound Function

When the `HoldSound` function is used, sound playback priority processing differs from normal processing. Assuming that the priority value set is 64, when playback starts the priority is 63 (one less than 64). If playback is successful, then the priority is set to the value of 64.

Normally the latter of two processes with the same priority takes precedence, but for this reason, when `HoldSound` is used the former takes precedence.

3.8 Playback Using the PrepareSound Function

Depending on the sound data type, playback may not start immediately even if the `StartSound` function is called. For example, when playing a stream, because some data must be pre-loaded, playback does not begin until loading completes.

Normally, this is not a major issue, but there are cases when a problem can arise if the timing of the start of playback is inconsistent, such as when synchronizing images and sound. In these cases, the `PrepareSound` function can be used in place of the `StartSound` function.

3.8.1 Using the PrepareSound Function

The `PrepareSound` function performs only the preparation for starting sound playback. For this reason, sound will not be played by calling only the `PrepareSound` function.

Preparation for starting sound playback occurs asynchronously. Use the `nw::snd::SoundHandle::IsPrepared` function to verify whether preparations have completed. After confirming that preparations are finished, start sound playback by calling the `nw::snd::SoundHandle::StartPrepared` function.

If preparations are complete, call the `StartPrepared` function to start sound playback immediately. If the `StartPrepared` function is called before preparations are complete, sound playback will begin automatically after waiting for preparations to complete.

3.9 Playback Using a String

In the sample code described previously, the sound ID of the sound to be played by the playback function was passed in the following ways.

Code 3-14 Playback Using a Sound ID

```
bool result = m_ArchivePlayer.StartSound(
      &m_Handle, SEQ_MARIOKART ).IsSuccess();
```

You must include a sound ID file (`*.csid`) ahead of time because this sound ID is defined in a sound ID file. You must therefore recompile each time the sound ID file is updated.

In addition to the above, a method of playing back sounds using a string is also provided.

Code 3-15 Playback Using a String

```
bool result = m_ArchivePlayer.StartSound(
      &m_Handle, "SEQ_MARIOKART" ).IsSuccess();
```

A sound ID file does not need to be included in this case because a string is used under this method. Note, however, that it is necessary to add the following type of initialization.

Code 3-16 Loading Label String Data

```
// Load STRING block
{
    u32 stringBlockSize = m_Archive.GetLabelStringDataSize();
    m_pMemoryForStringBlock = MemAlloc( stringBlockSize );
    if ( ! m_Archive.LoadLabelStringData(
          m_pMemoryForStringBlock, stringBlockSize ) )
    {
        NW_ASSERTMSG( 0, "cannot load stringBlock(%s)", SOUND_ARC_PATH );
    }
}
```

Playback using a string has the following characteristics in comparison to playback using a sound ID. Use string playback according to your needs.

- Because there is no sound ID file, programs do not need to be recompiled even if sound data is updated.
- It is necessary to load label string data.
- There is execution cost due to converting the string into a sound ID.
- Spelling mistakes in sound specifications cannot be detected during compile.

Be sure to refer to the sample demo titled `labelString`, which provides an example of playback using a string.

3.10 Types of Sounds

There are three different types of sounds that can be played with the nw::snd Library:

- Stream sounds
- Wave sounds
- Sequence sounds

All of these sounds can be played using the nw::snd::SoundArchivePlayer::StartSound function. Thus, the programmer does not need to be concerned about which types of sounds were created by the sound designer. Moreover, functions are prepared for all of the basic operations of the sound handle (for example, pausing, stopping, and changing the volume and pitch) so common operations can be used to control playback.

To perform operations that can only be performed on sequence sounds, such as changing the tempo, use a sequence sound handle. Sequence sound handles are treated the same as regular sound handles, but they have additional functions specific to sequence control.

To use a sequence sound handle, pass the sound handle to use as an argument and call the nw::snd::SequenceSoundHandle class constructor. Then call functions for specific sequence operations.

Code 3-17 Using Sequence Sound Handles

```
if ( m_ArchivePlayer.StartSound( &m_Handle, SEQ_MARIOKART ).IsSuccess() )
{
    nw::snd::SequenceSoundHandle seqHandle( &m_Handle );
    seqHandle.SetTempoRatio( 2.0f );
}
```

If a handle bound to a sound other than a sequence sound is passed to an nw::snd::SequenceSoundHandle class constructor, that sequence sound handle will be disabled. Function calls for disabled sequence sound handles are ignored.

4 Memory Management

Memory management plays a very small role in the `simple` demo. The only time memory management is involved in this demo is when the heap is built during initialization and later, when the data is loaded to the heap.

This chapter provides information about managing memory.

4.1 How to Use the Sound Heap

Since the sound heap class `nw::snd::SoundHeap` is a stack-type heap (LIFO), memory space is reserved in order from the top and released in order from the bottom. Memory from the heap is automatically reserved when sound data is loaded. This process is carried out by the `nw::snd::SoundDataManager::LoadData` function.

Sound data that is no longer needed is deleted by releasing the corresponding region of memory. There are two ways to release the memory region:

- Clear all
- Restore prior state

4.1.1 Clear All

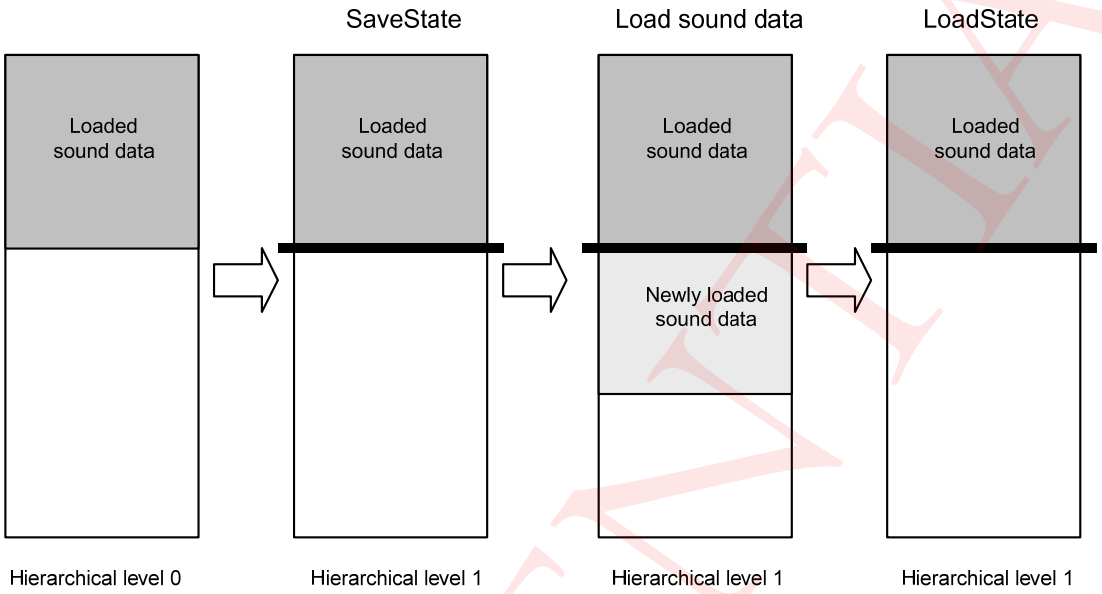
This deletes all sound data and restores the initial state. To clear all sound data in the sound heap, call the `nw::snd::SoundHeap::Clear` function.

4.1.2 Restore Prior State

Use this method to delete only the unnecessary data. First, save the current state by calling the `nw::snd::SoundHeap::SaveState` function. As the return value, this function returns the hierarchical level, which indicates the state after saving. You can then use this value to restore the state to the state of the saved heap.

If, after loading a number of sets of sound data, you call the `nw::snd::SoundHeap::LoadState` function using this above-mentioned value for the hierarchical level, you can return to the state that existed immediately after the call to `nw::snd::SoundHeap::SaveState`. This has the effect of deleting all data that was loaded after the call to `nw::snd::SoundHeap::SaveState`.

Figure 4-1 The LoadState Function Can Be Used to Restore a Prior State



At this point, you can continue playing the sound that was being played using the loaded sound data. The `nw::snd::SoundHeap::SaveState` function can be called repeatedly, and each time the hierarchical level increases in value.

4.2 Sound Heap Application

4.2.1 Feature to Automatically Stop Invalid Sound

When using data on the sound heap to playback sound, if the region where that data is stored is released, the playback of that sound is automatically stopped. Therefore, it is unnecessary for the application to confirm whether the sound being played has stopped before a region is released.

However, because the sound is automatically stopped immediately, it may sound unnatural. In these cases, have the application apply a fade-out before releasing the data storage region.

4.2.2 Multiple Sound Heaps

Normally only a single sound heap is created, but it is also possible to create multiple sound heaps. By preparing multiple heaps you can save and restore the state of each heap independently.

To use multiple sound heaps, all you need to do is create multiple instances of `nw::snd::SoundHeap` and build the heaps using the `nw::snd::SoundHeap::Create` function. You always need to pass a sound heap to the `nw::snd::SoundDataManager::LoadData` function, so this gives you a way to specify the sound heap from which to reserve the memory.

4.2.3 SoundMemoryAllocatable Class

Up to this point, it has been explained that the sound heap must be passed to `nw::snd::SoundDataManager::LoadData`, but strictly speaking it is the `nw::snd::SoundMemoryAllocatable` class and not the `nw::snd::SoundHeap` class that must be passed. However, because the `nw::snd::SoundHeap` class inherits the `nw::snd::SoundMemoryAllocatable` class, it can be passed as is.

The `nw::snd::SoundMemoryAllocatable` class is an interface class where the `Alloc` function is defined as a purely virtual function. By having the application independently implement the `Alloc` class, memory can be secured from heaps outside of the `nw::snd::SoundHeap` class.

The `originalSoundHeap` demo is provided for reference and shows a sample of an independent `Alloc` class that inherits from the `nw::snd::SoundMemoryAllocatable` class.

However, when the `nw::snd::SoundHeap` class is not used, be aware that the “Feature to Automatically Stop Invalid Sound” does not work. In this case, if the sound data being played is mistakenly discarded, unexpected sound output may occur.

4.3 Player Heap

4.3.1 The Player Heap

In addition to the sound heap, a player heap is also available.

The player heap temporarily stores data only for sound playback. It is automatically allocated when sound playback begins and automatically deallocated when playback stops. In addition to sequence data, all sound data can be loaded in the player heap.

4.3.2 Using the Player Heap

Because the allocating and deallocating of the player heap occurs automatically in the library, the programmer does not need to manage the player heap. In addition, the sound archive created by the sound designer determines whether the player heap is used.

When sound is played using the player heap, the data load occurs automatically. The data load process occurs asynchronously with the sound data load thread created with the `nw::snd::SoundSystem::Initialize` function and not with a thread called by a sound playback start function (`nw::snd::SoundArchivePlayer::StartSound`, and so on). Accordingly, sound does not start to play immediately after calling the sound playback start function, but sound automatically starts playing after the data load completes. To control the timing when sound starts to play, the `PrepareSound` function can be used in place of the `StartSound` function. For details on the `PrepareSound` function, see section 3.8 Playback Using the PrepareSound Function.

4.3.3 Memory for the Player Heap

The memory region for the player heap is allocated with the `nw::snd::SoundArchivePlayer::Initialize` function. The information for how much memory to allocate to the player heap is embedded in the sound archive created by the sound designer. For player heaps, exactly the maximum number of sound playback times for that player will be allocated.

The specific memory size can be obtained with the `nw::snd::SoundArchive::ReadPlayerInfo` function.

4.3.4 Cautions for Playback with Different Players

Although sound can be played on a player directly specified by the API without using the player set by the sound data, exercise caution when using the player heap.

The player heap is managed separately for each player, so the existence of a player heap and its size differs by player. Accordingly, if a sound using a player heap is played on a player that either does not have a player heap or has a player heap that is too small, playback will fail.

5 Sound Actors

This chapter describes using sound actors that manage sound for each actor.

5.1 Sound Actors

Some characters in a game may have audible footsteps as they walk or talk as they swing a sword. At this time, the sounds of the footsteps, voice, and sword are made by a single character. Sound actors collectively manage the multiple sounds produced by a single character.

The following are possible when using sound actors.

- When a character disappears from a game, all of the sounds that the character makes can be stopped together.
- All of the parameters, such as volume, for the sounds produced by a character can be changed at once.
- The number of sounds that one character can make simultaneously can be limited.

5.2 Executing the Sample Demo

First, run the `soundActor` demo that uses sound actors.

5.2.1 Starting the Sample Demo

The sample demo is stored in the `$NW4C_ROOT/Library/demos/snd/soundActor` directory, so start it in the same manner as the `simple` demo.

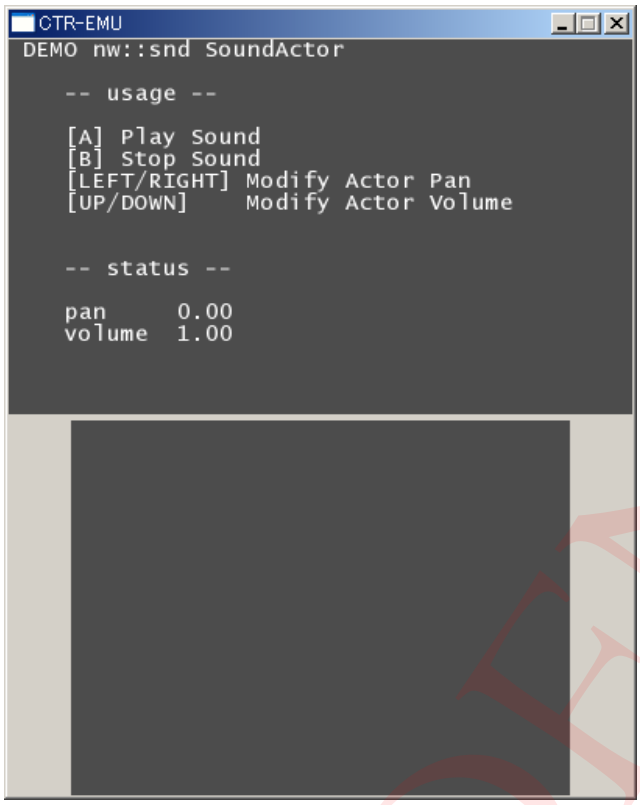
5.2.2 Controls

Control the demo using either the PC keyboard (for the PC version) or the CTR-TEG2 buttons (for the CTR-TEG2 version). Button assignments are as shown in Table 5-1.

Table 5-1 `soundActor` Demo Button Assignment

Button	Description
A Button	Start sound playback
B Button	Stop sound playback
Left and Right on the +Control Pad	Change actor panning
Up and Down on the +Control Pad	Change actor volume

Figure 5-1 soundActor Demo Screens



5.3 The SoundActor Program

This section describes the sound program and the source code.

The source code is in the SoundActor.h file and the SoundActor.cpp file in the $NW4C_ROOT/Library/demos/snd/soundActor/sources directory.

5.3.1 Initializing the Instances

The program creates instances of and initializes the sound actor (SoundActor) class.

Code 5-1 SoundActor Instance Initialization

```
// SoundActorApp.h
nw::snd::SoundActor m_Actor;

// SoundActorApp.cpp
m_Actor.Initialize( m_ArchivePlayer );
```

A reference to the sound archive player (nw::snd::SoundArchivePlayer) is passed as an argument of the nw::snd::SoundActor class Initialize function. When

nw::snd::SoundActor is used to playback sound, the result is that the nw::snd::SoundArchivePlayer is used for playback.

5.3.2 Sound Playback Using Sound Actors

To playback sound using sound actors, use nw::snd::SoundActor in place of nw::snd::SoundArchivePlayer for playback. nw::snd::SoundActor has the same playback functions as nw::snd::SoundArchivePlayer.

Code 5-2 Sound Playback Using Sound Actors

```
bool result = m_Actor.StartSound( &m_Handle, SE_YOSHI ).IsSuccess();
NN_LOG("SE_YOSHI ... (%d)¥n", result);
```

5.3.3 Setting the Maximum Number of Sounds to Be Played Simultaneously

The maximum number of sounds that can be played per actor can be set. The default is an unlimited number of sounds.

Code 5-3 Setting the Maximum Number of Sounds to Be Played Simultaneously

```
m_Actor.SetPlayableSoundCount( 0, 1 );
```

The first argument is the actor player number. The actor player is described in detail below.

The second argument is the maximum number of sounds that can be played simultaneously. In this example, a limit of only one sound to be played simultaneously is applied.

5.3.4 Parameter Updates

The volume and pan values for each actor can be changed.

Code 5-4 Parameters Updates

```
m_Actor.SetVolume( m_ActorVolume );
m_Actor.SetPan( m_ActorPan );
```

5.3.5 Deallocating Sound Actors

When a nw::snd::SoundActor instance is destroyed, such as when an actor is removed, the sound played by that actor continues to play. To stop the sound, the nw::snd::SoundActor::StopAllSound function must be called explicitly.

5.4 Actor Players

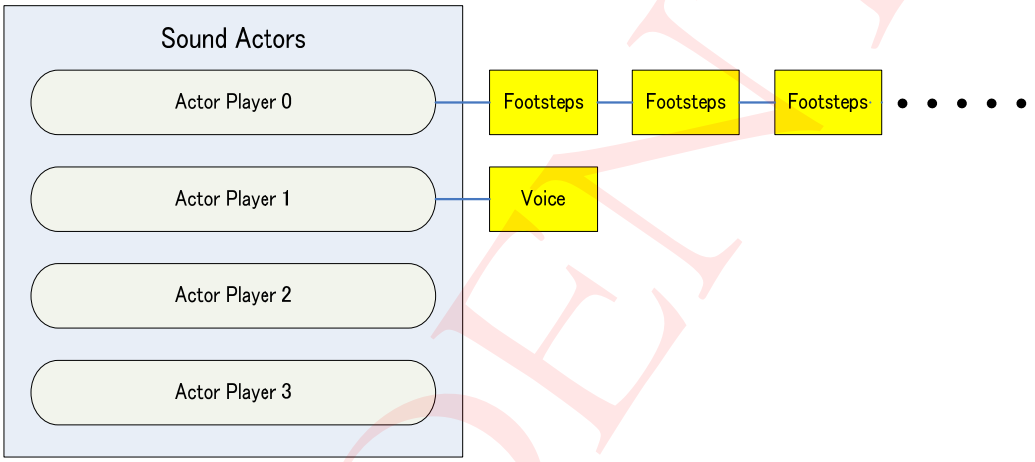
5.4.1 Maximum Number of Simultaneous Sounds Played and the Actor Player

Although it was explained that the maximum number of sounds that are played simultaneously can be set for each sound actor, more precisely, the maximum can be set for each actor player.

Each sound actor has four actor players. A maximum number of sounds that are played simultaneously can be set for each actor player, from number 0 to number 3.

For example, if there is a character that has footsteps and speaks, there are cases when it is desirable to limit the voice to one and to have an unlimited number of footsteps. For this case, set only actor player number 1 to a maximum number of sounds that are played simultaneously to one and set the voice to playback with actor player number 1. Do not set a limit for actor player number 0, and set the footsteps to play back on actor player number 0.

Figure 5-2 Actor Player



When a sound actor instance is created, the maximum number of sounds to be played simultaneously is set to unlimited for actor player number 0, but is limited to one each for actor player numbers 1, 2, and 3.

5.4.2 Setting the Number of the Actor Player to Be Played

Set the number of the actor player to be played in the sound data as a parameter for each sound. The default setting is actor player number 0.

This can be set in the **Actor Player** column in the SoundMaker program.

5.5 Dynamically Changing the Sound ID

If a sound actor is used, the sound being played can change dynamically according to the actor status.

For example, when footsteps are being played, it may be desirable to change the sound being played according to the type of surface where the actor is walking. Normally, it is enough to change the sound ID passed in the arguments to the `StartSound` function, but you can also change the ID of the sound for playback by overriding the `nw::snd::SoundActor::SetupSound` function.

5.5.1 The SoundActor::SetupSound Function

The nw::snd::SoundActor::SetupSound function is a virtual function. You can inherit from the nw::snd::SoundActor class and then override this function.

Code 5-5 The SoundActor::SetupSound Function

```
virtual SoundStartable::StartResult SetupSound(
    SoundHandle* handle,
    u32 soundId,
    const SoundStartable::StartInfo* startInfo,
    void* setupArg
);
```

The nw::snd::SoundActor::SetupSound function is called when playing back sounds. The arguments are mostly the same as for the StartSound function. The fourth argument setupArg takes the parameters needed for sound setup. When calling the SetupSound function of the base class, this argument must be passed without being altered.

5.5.2 Changing the Sound ID

Here, we use the MySoundActor class, which inherits from the nw::snd::SoundActor class, to show how to change the sound of jumping on the ground depending on the type of ground. By way of example, we implement the MySoundActor::SetupSound function.

Code 5-6 Changing the Sound ID

```
nw::snd::SoundStartable::StartResult MySoundActor::SetupSound(
    SoundHandle* handle,
    u32 soundId,
    const SoundStartable::StartInfo* startInfo,
    void* setupArg
)
{
    if ( soundId == BOUND1 )
    {
        switch ( m_FloorType )
        {
        case FLOOR_TYPE_A:
            soundId = BOUND1;
            break;
        case FLOOR_TYPE_B:
            soundId = BOUND2;
            break;
        case FLOOR_TYPE_C:
            soundId = BOUND3;
```

```
            break;
        }
    }

    return nw::snd::SoundActor::SetupSound(
        handle,
        soundId,
        startInfo,
        setupArg
    );
}
```

6 3D Sound

This section describes using 3D Sound that changes the volume and pan of sound according to 3D spatial coordinates.

6.1 3D Sound Structure

The 3D sound structure is comprised of the following three classes:

- The 3D Sound Manager (`Sound3DManager`)
- The 3D Sound Actor (`Sound3DActor`)
- The 3D Sound Listener (`Sound3DListener`)

6.1.1 The 3D Sound Manager (Sound3DManager)

The 3D sound manager calculates and manages 3D sound parameters. Basically, one instance is generated and used.

6.1.2 The 3D Sound Actor (Sound3DActor)

The 3D sound actor represents a single sound source. It sets 3D spatial coordinates and manages sound played by a sound source.

When playing back with 3D sounds, sound is played using a 3D sound actor instead of with the sound archive player (`SoundArchivePlayer`).

The `Sound3DActor` class inherits the `SoundActor` class.

6.1.3 The 3D Sound Listener (Sound3DListener)

The 3D sound listener represents a microphone or human ear. To set the listener position and orientation, a transformation matrix called the listener matrix is used.

6.2 Sample Demo Execution

Begin by running the 3DSound sample demo that uses `sound3D`.

6.2.1 Start the Sample Demo

The sample demo is stored in the `$NW4C_ROOT/Library/demos/snd/sound3d` directory, so start it in the same manner as the `simple` demo.

6.2.2 Controls

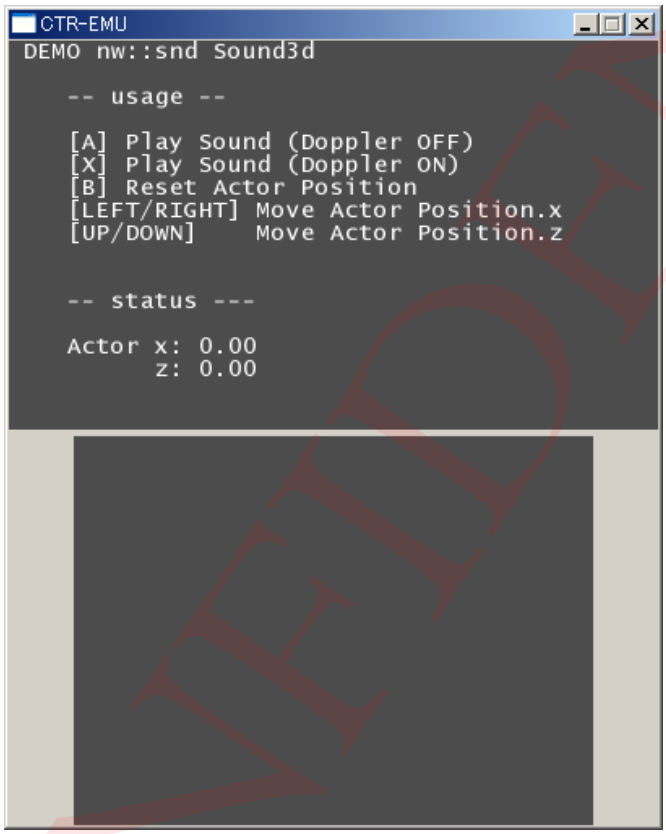
Use either the PC keyboard (for the PC version) or the CTR-TEG2 buttons (for the CTR-TEG2) to control the demo.

Button assignments are shown in Table 6-1.

Table 6-1 sound3d Demo Button Assignment

Button	Description
A Button	Start sound playback (no Doppler effect)
X Button	Start sound playback (with Doppler effect)
B Button	Reset actor coordinates
Left and Right on the +Control Pad	Change actor X coordinate
Up and Down on the +Control Pad	Change actor Z coordinate

Figure 6-1 sound3d Demo Screens



6.3 3D Sound Program

In this section, the sound program is described along with the source code.

The source code is in the `Sound3dApp.h file` and `Sound3dApp.cpp` file in the `$NW4C_ROOT/Library/demos/snd/sound3d/sources` directory.

6.3.1 Creating Instances

Instances of the three classes described above are created.

Code 6-1 Instance Initialization

```
nw::snd::Sound3DManager    m_3dManager;
nw::snd::Sound3DActor      m_3dActor;
nw::snd::Sound3DListener   m_3dListener;
```

Normally, only one instance each of `nw::snd::Sound3DManager` and `nw::snd::Sound3DListener` is created, and a `nw::snd::Sound3DActor` instance must be created for each sound source.

6.3.2 Initializing the 3D Sound Manager

The `nw::snd::Sound3DManager` must be initialized as shown in the following code.

Code 6-2 3D Sound Manager Initialization

```
// Initialize 3D sound manager
{
    size_t setupSize = m_3dManager.GetRequiredMemSize( &m_Archive );
    m_pMemoryFor3dManager = MemAlloc( setupSize );
    m_3dManager.Initialize( &m_Archive, m_pMemoryFor3dManager, setupSize );
    m_3dManager.SetMaxPriorityReduction( 32 );
    m_3dManager.SetSonicVelocity( 340.0f / 60 );
}
```

See section 6.4 Doppler Effect for details on the `nw::snd::Sound3DManager::SetSonicVelocity` function.

6.3.2.1 Initializing

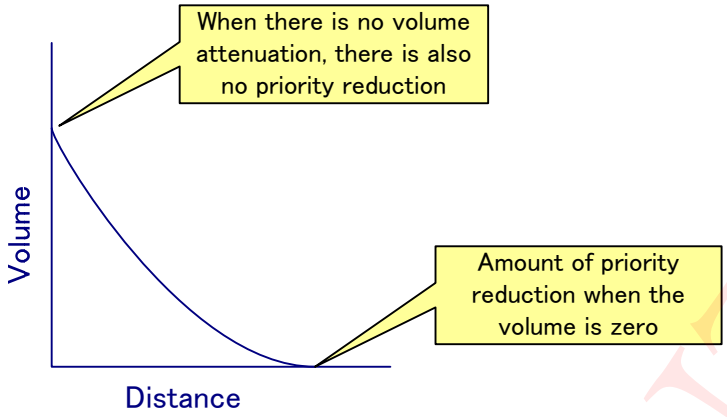
The `nw::snd::Sound3DManager::Initialize` function initializes the 3D sound manager. The arguments take a pointer to the sound archive being used and the main memory region required for initialization. The required memory region size can be obtained using the `nw::snd::Sound3DManager::GetRequiredMemSize` function.

6.3.2.2 Maximum Priority Reduction

The maximum priority reduction is set using the `nw::snd::Sound3DManager::SetMaxPriorityReduction` function.

The priority of the sound being played as a 3D sound normally decreases in relation to the drop in volume. The maximum priority reduction represents the priority reduction when the volume drops to 0.

Figure 6-2 Reducing the Priority Value



6.3.3 Initializing the Listener

The `nw::snd::Sound3DListener` function must be initialized as shown in Code 6-3.

Code 6-3 Initializing the Listener

```
// Initialize 3D sound listener
{
    m_3dManager.AddListener( &m_3dListener );

    CalcListenerMatrix( &m_ListenerMtx );
    m_3dListener.SetMatrix( m_ListenerMtx );
    m_3dListener.SetMaxVolumeDistance( 5.0f );
    m_3dListener.SetUnitDistance( 5.0f );
    m_3dListener.SetInteriorSize( 5.0f );
}
```

6.3.3.1 Listener Registration

Register the listener in the 3D sound manager using the `nw::snd::Sound3DManager::AddListener` function.

Multiple listeners may also be registered in a single 3D sound manager. For details, see section 6.5 Multi-Listeners.

6.3.3.2 The Listener Matrix

Set the listener matrix using the `nw::snd::Sound3DListener::SetMatrix` function.

In the sample demo, the listener matrix is determined using the following procedure.

Code 6-4 Calculating the Listener Matrix

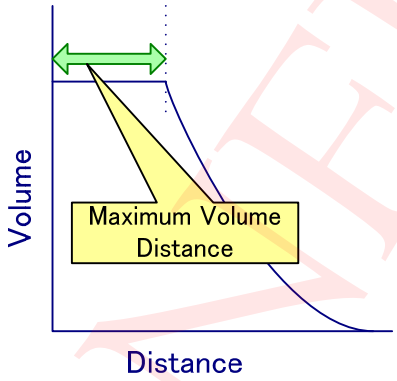
```
void Sound3dApp::CalcListenerMatrix( nw::math::MTX34* mtx )
{
   const nw::math::VEC3 pos( 0.0f, 0.0f, -3.0f );   // Listener position
   const nw::math::VEC3 upVec( 0.0f, 1.0f, 0.0f );  // Listener Up vector
   const f32 degree = 0.0f;    // Listener facing

   // Listener directional vector
   const nw::math::VEC3 direction(
      -nw::math::SinDeg( degree ), 0.0f, -nw::math::CosDeg( degree )
   );
   nw::math::VEC3 target = pos + direction;

   // Generate listener matrix
   nw::math::MTX34LookAt( mtx, &pos, &upVec, &target );
}
```

6.3.3.3 Maximum Volume Distance

The maximum volume distance is set with the `nw::snd::Sound3DListener::SetMaxVolumeDistance` function.

The maximum volume distance is the distance within which the volume remains at its maximum level, even if the position changes. It is specified as a distance from the listener position.

Figure 6-3 Maximum Volume Distance



By setting the maximum volume distance, you can prevent the volume from changing dramatically when an actor's position changes near the listener.

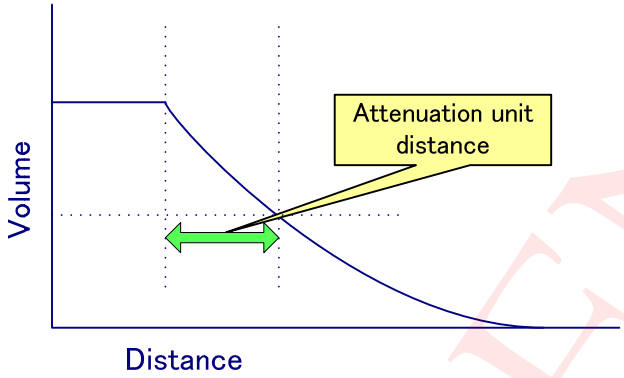
6.3.3.4 Attenuation Unit Distance

The attenuation unit distance is set using the `nw::snd::Sound3DListener::SetUnitDistance` function.

The attenuation unit distance indicates the distance at which the volume falls to approximately half its value. Because the maximum volume distance is not included in this distance, the volume falls to half when the actor moves to a distance from the listener that is (maximum volume distance) + (attenuation unit distance).

Although volume falls to half at the attenuation unit distance by default, the volume attenuation rate at the attenuation unit distance can be changed for each separate sound by manipulating the sound data.

Figure 6-4 Attenuation Unit Distance



6.3.3.5 Interior Size

The interior size is set with the `nw::snd::Sound3DListener::SetInteriorSize` function.

The interior size is the size of the area in which the pan and surround pan vary. It is set as a (radial) distance from the listener's position.

If the interior size is large, pan changes become smooth. Conversely, if the interior size is small, pan changes become sudden. For details, see the *Sound System Manual*.

Note: Although sound volume is attenuated as the distance from the listener increases, the amount of attenuation depends on the settings made for the maximum sound volume range (described above).

For instance, if a sound that is played first at the listener position is moved to the right, the sound should play first from all the speakers at equal volume and then shift to the right. When the sound reaches the interior size position, it should play only from the right speaker. Even if it moved further to the right, the pan would not change (but the volume should attenuate based on the distance).

6.3.4 Initialize the Actor

`nw::snd::Sound3DActor` must be initialized as shown in the following code.

Code 6-5 Initialize the Actor

```
// Initialize 3D sound actor
{
    m_3dActor.Initialize( m_ArchivePlayer, m_3dManager );
    m_ActorPos = nw::math::VEC3::Zero();
    m_3dActor.SetPosition( m_ActorPos );
}
```

The arguments to the `nw::snd::Sound3DActor::Initialize` function take references to a sound archive player (`nw::snd::SoundArchivePlayer`) and a 3D sound manager (`nw::snd::Sound3DManager`). Playing back a sound using the `nw::snd::Sound3DActor` function effectively plays back the sound using the `nw::snd::SoundArchivePlayer` object passed in the arguments. Similarly, the 3D sound parameters for the sound being played back are calculated from the `nw::snd::Sound3DManager` object passed in the arguments.

6.3.4.1 Actor Coordinates

The actor coordinates are set with the `nw::snd::Sound3DActor::SetPosition` function.

6.3.5 3D Sound Playback

All 3D sounds are played back using `snd::Sound3DActor` rather than using `nw::snd::SoundArchivePlayer`. The `snd::Sound3DActor` class has a sound playback function similar to `nw::snd::SoundArchivePlayer`.

Code 6-6 3D Sound Playback

```
m_3dActor.HoldSound( &m_Handle, SE_SQUARE );
```

6.3.6 Updating the Actor’s Coordinates

Updating the actor’s coordinates as needed will also update the 3D parameters of the sound being played.

Code 6-7 Updating Actor Coordinates

```
m_3dActor.SetPosition( m_ActorPos );
```

The listener matrix can be updated using the same method as the actor’s coordinates.

6.3.7 Lifespans of Actors and Sounds

Even if an instance of `nw::snd::Sound3DActor` is destroyed (for example, when an actor disappears), the sound that was played by the actor will continue to play. Also, because position information is stored on the sound side, 3D sound parameters will be calculated correctly so that the sound will play at the position where the actor disappeared, even if the listener matrix is updated.

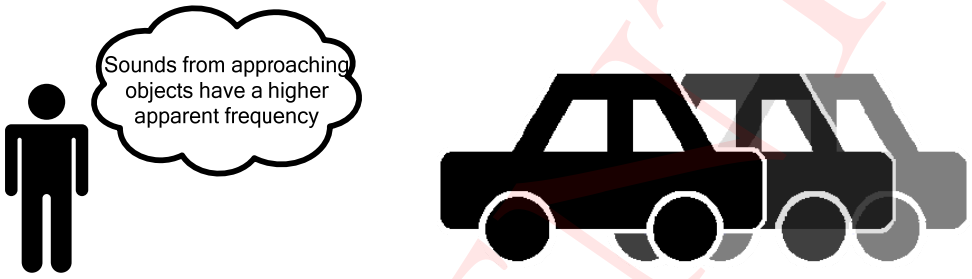
Accordingly, programmers do not need to worry about lifetimes of `nw::snd::Sound3DActor` instances.

For example, if position data is not updated continuously, the instance of `nw::snd::Sound3DActor` could be destroyed right after playing the sound.

6.4 Doppler Effect

The following describes the method of representing the Doppler effect, where the pitch of a sound depends on the relative velocity of the object producing the sound and the person hearing it.

Figure 6-5 Doppler Effect



The `sound3d` sample demo is provided as a reference for the Doppler effect.

6.4.1 Doppler Effect Parameters

The following two parameters must be set to apply the Doppler effect.

- Speed of sound
- Doppler factor

6.4.1.1 Speed of Sound

The speed of sound must be set to calculate the Doppler effect.

Code 6-8 Setting the Speed of Sound

```
m_3dManager.SetSonicVelocity( 340.0f / 60 );
```

The default value is 0.0f, so the Doppler effect is not applied to this value.

The unit for the value is the speed of sound per frame. The speed of sound is approximately 340 m/sec. So if it is assumed that 1.0f of the coordinates set by the `Sound3DActor::SetPosition` function is 1 meter, then if motion is over 60 frames, the value that should be set is 340.0f/60.

6.4.1.2 Doppler Factor

The Doppler factor is a parameter to adjust the amount of Doppler effect to apply.

The Doppler factor is set as a parameter for each sound in the sound data. The default value is 0, but this value can be increased to 127 to apply the Doppler effect. A standard value is 32.

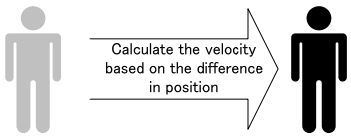
This can also be set using the **Doppler Effect** column in the SoundMaker program.

6.4.2 Changing the Tone

When the two parameters above are set, the Doppler effect is automatically applied.

Specifically, in either the `Sound3DActor::SetPosition` or `Sound3DListener::SetMatrix` functions, speed is calculated with the difference from the previous call, and then the Doppler effect is applied. It is unnecessary for the speed to be set by the application.

Figure 6-6 Automatic Calculation of Speed



6.4.3 Explicitly Specifying Speed

As mentioned above, speed is automatically calculated. But there may be cases where speed needs to be explicitly set by the application.

For example, if the camera position jumps to another location, a sudden pitch change may occur because the speed is calculated by the distance jumped and processing assumes an extreme speed. In this case, after the `Sound3DListener::SetMatrix` function is called, the speed must be explicitly specified with the `Sound3DListener::SetVelocity` function.

In addition, when generating a `Sound3DActor` that already has speed, the speed must be explicitly specified.

6.4.4 Formula for Pitch Change

The amount that pitch is changed is found with the following formula.

$$pitch = \frac{Vs - Vl \times F}{Vs - Va \times F}$$

Term	Definition
`pitch`	Ratio of pitch change
`Vs`	Speed of sound
`Vl`	Speed of the listener
`Va`	Speed of the actor
`F`	Doppler Factor ÷ 32

6.5 Multi-Listeners

Multiple listeners can be registered in a single 3D sound manager. The operational specifications are different for one listener and for multiple listeners, so to distinguish between the two, a single listener is called a single listener and multiple listeners are called multi-listeners.

Multi-listeners are used when the game screen is divided and displayed in several sections. When the screen is divided, several cameras come into existence, so multiple listeners must also be registered.

6.5.1 Differences Compared to a Single Listener

A multi-listener differs in the following ways from a single listener.

- The volume takes the maximum value of the results calculated for each listener.
- The priority is the highest priority value of the results calculated for each listener.
- Pan and surround pan are not changed.
- The Doppler effect cannot be applied.

6.6 3D Sound Customization

Up to this point, the standard operations of 3D sound have been described, but many of these can be freely customized. For example, the volume attenuation curve due to distance can be either logarithmic or linear, but can be customized to any curve.

6.6.1 3D Sound Engines (Sound3DEngine)

The calculation processing for 3D sound is performed by a 3D sound engine. One 3D sound engine is allocated to the 3D sound manager. The 3D sound engine implements a default 3D sound calculation process, but a proprietary 3D sound calculation process can be performed by customizing it.

6.6.2 Creating a 3D Sound Engine

To create a proprietary 3D sound engine, create a class that inherits the `Sound3DEngine` class.

Code 6-9 3D Sound Engine Inheritance

```
class MySound3DEngine : public nw::snd::Sound3DEngine
{
  ...
}
```

The created `MySound3DEngine` class instance is registered in `Sound3DManager`.

Code 6-10 3D Sound Engine Registration

```
MySound3DEngine m_3dEngine;

Sound3DManger::SetEngine( &m_3dEngine );
```

6.6.3 Implementing 3D Sound Calculation Processing

In order to employ 3D sound calculation processing, the `Sound3DEngine::UpdateAmbientParam` function must be overridden.

Code 6-11 Overriding the UpdateAmbientParam Function

```
class MySound3DEngine : public nw::snd::Sound3DEngine
{
   virtual void UpdateAmbientParam(
       const nw::snd::Sound3DManager* manager,
       const nw::snd::Sound3DParam* actorParam,
       u32 soundId,
       u32 updateFlag,
       nw::snd::SoundAmbientParam* param
   );
}
```

This function is periodically called to update the 3D sound parameters. If this function is called, the content of the `SoundAmbientParam` structure (the last argument) is updated as needed.

Each argument is described in the sections below.

6.6.3.1 Sound3DManager

The registered listener list can be gotten from `Sound3DManager`. 3D sound calculation is based on this listener information.

6.6.3.2 Sound3DParam

The `Sound3DParam` structure stores the `Sound3DActor` coordinates and the 3D sound related parameters set for each sound. When 3D sound is calculated, these parameters must be reflected.

6.6.3.3 Sound ID

The Sound ID can be used as a hint for 3D sound calculation. It can be ignored if it is not needed.

6.6.3.4 Update Flag

This is a bit flag that indicates the parameters that should be updated. Only the parameters with a valid bit are updated.

6.6.3.5 SoundAmbientParam

The `SoundAmbientParam` structure includes the following members.

Code 6-12 The `SoundAmbientParam` Structure

```
struct SoundAmbientParam
{
  f32 volume;
  f32 pitch;
  f32 pan;
```

```
  f32 span;
  f32 fxSend;
  f32 lpf;
  f32 biquadFilterValue
  int biquadFilterType
  int priority;
  u32 userData;
  ...
```

These structure members are updated based on the data from the four arguments mentioned above. The `fxSend` value specified here is reflected in AUX bus A.

Note: In the current implementation, the only valid members are `volume`, `pitch`, `pan`, `priority`, and `userData`.

6.6.4 Sound3DCalculator

Because 3D sound calculations are complex, the `Sound3DCalculator` utility class has been prepared to allow for easier customization.

Code 6-13 Sound3DCalculator Class

```
class Sound3DCalculator
{
  public:
    static void CalcVolumeAndPriority(
        const Sound3DManager& manager,
        const Sound3DListener& listener,
        const Sound3DParam& actorParam,
        f32* volumePtr,
        int* priorityPtr
    );
    static void CalcPan(
        const Sound3DManager& manager,
        const Sound3DListener& listener,
        const Sound3DParam& actorParam,
        const CalcPanParam& calcPanParam,
        f32* panPtr,
        f32* spanPtr
    );
    static void CalcPitch(
        const Sound3DManager& manager,
        const Sound3DListener& listener,
        const Sound3DParam& actorParam,
        f32* pitchPtr
    );
```

If you use the `Sound3DCalculator` class, you can calculate the 3D sound parameters using the `UpdateAmbientParam` function arguments almost without modification. Reference the implementation code of the `Sound3DCalculator` class for an easy and more detailed customization.

Note: NW4R included a multi-voice output feature, but due to the amount of processing required for multiple output voices, this has not been implemented in NW4C.

7 Notes About Using the nn::snd Library

The nw::snd Library is implemented using the CTR_SDK's nn::snd Library. For this reason, you need to keep the following points in mind when your application makes direct use of the nn::snd Library.

To review actual examples of the library in use, see the `withSdk` sample demo and the `createSoundThreadManually` sample demo.

7.1 nn::snd::AllocVoice

When the application is using the nn::snd Library (the same as only using the CTR_SDK), the `nn::snd::AllocVoice` function can get the voice (`nn::snd::Voice`). Furthermore, all operations are allowed on the obtained voice.

However, the `nw::snd` Library makes an internal call to the `nn::snd::AllocVoice` function to try to get the voice, so if the application will be allocating voices, you will need to limit the number of voices used by the library or introduce some other mechanism, such as giving high priority to the getting of voices.

To limit the number of voices used by the `nw::snd` Library, call the `nw::snd::SoundSystem::SetMaxVoiceCount` function. For example, if you know that the application will use four voices, describing the following code will limit the number of voices used by the `nw::snd` Library.

Code 7-1 nw::snd Limiting Number of Voices Used By Library

```
nw::snd::SoundSystem::SetMaxVoiceCount( NN_SND_VOICE_NUM - 4 );
```

When stream sounds are played by the `nw::snd` Library, voices are always obtained according to the priority of `nn::snd::VOICE_PRIORITY_NODROP`. When other sounds are being played (i.e., wave sounds and sequence sounds) the voices are obtained with priority lower than that (depending on such factors as the channel priority).

7.1.1 Precautions When Threads Are Running in the System Core

To run a sound thread in the system core, you must set the priority ahead of time to `nn::snd::VOICE_PRIORITY_NODROP`. (See Chapter 8, Operations in the System Core.)

If a priority other than this is specified, the **`nn::snd::AllocVoice`** function will stop execution on an assert. (This is true in the case of Debug and Development versions only. For Release versions, the assert is ignored and execution does not stop, but operations are not guaranteed.)

Note: As mentioned above, if you get a voice for a wave sound or sequence sound with a priority lower than `nn::snd::VOICE_PRIORITY_NODROP`, the number of sounds that can be played by the nw::snd library will go down if too many are independently obtained using `nn::snd::VOICE_PRIORITY_NODROP.`

7.1.2 Synchronous Processing

As of NW4C-1.2.15, when performing operations on a voice independently allocated by the **nn::snd::AllocVoice** function must be synced with the sound thread using the API functions listed below regardless of the sound thread's operating core.

- The **nw::snd::SoundSystem::LockSoundThread** function and the **nw::snd::SoundSystem::UnlockSoundThread** function, or
- Functions in the **nw::snd::SoundSystem::SoundThreadScopedLock** class

The phrase "performing operations on voices" refers to calling any of the following API functions provided by CTR_SDK.

- All `nn::snd::Voice` class functions
- The `nn::snd::AllocVoice` function
- The `nn::snd::FreeVoice` function

Any thread may call these functions on voices. However, if a sound thread is operating in the system core and one of the above functions is called on a voice independently allocated from inside that sound thread, there is no need to synchronize operations using the API functions used for synchronization given above. (See the Note below regarding executing some sort of process inside the sound thread.)

Note: The term "sound thread" refers to a thread that calls the nn::snd::WaitForDspSync and nn::snd::SendParameterToDsp functions provided by CTR_SDK. Usually these two functions are called inside the nw::snd library (inside the CTR_SDK API for NW4C for CTR_SDK-013.0 support versions and later). You can execute a user process inside a sound thread by using the **nw::snd::SoundSystem::SetSoundFrameUserCallback** function or created the thread that will call the two functions above without the nw::snd library creating a sound thread. For descriptions of each of these methods, see Section 7.3.1, Using the nw::snd::SoundSystem::SetSoundFrameUserCallback callback functionand Section 7.2 nn::snd::WaitForDspSync, nn::snd::SendParameterToDsp.

7.2 nn::snd::WaitForDspSync, nn::snd::SendParameterToDsp

The application is prohibited from calling the functions `nn::snd::WaitForDspSync` and `nn::snd::SendParameterToDsp` because they are called from sound threads generated and activated inside the `nw::snd` Library (inside the CTR_SDK API for NW4C for CTR_SDK-013.0 support versions and later)..

However if `nw::snd::SoundSystem::SoundSystemParam::autoCreateSoundThread` is set to `false`, then the `nw::snd` Library will not create and generate sound threads internally. When this is the case, the application is permitted to call the `nn::snd::WaitForDspSync` and `nn::snd::SendParameterToDsp` functions.

If the application is going to call these two functions, it must call the `nw::snd::SoundSystem::SoundFrameProcess` function between the two.

For details, see the `demos/snd/createSoundThreadManually` demo.

7.3 Matching the Timing of Parameter Settings

When the application allocates `nn::snd::Voice` instances, the calls to `nn::snd::Voice::SetVolume` and other functions that set parameters must be timed to take place between the call to the `nn::snd::WaitForDspSync` function and the call to the `nn::snd::SendParameterToDsp` function. (In the examples given below, this period of time is called *timing [1]*.)

There are two ways this can be implemented:

7.3.1 Use nw::snd::SoundSystem::SetSoundFrameUserCallback

The `nw::snd::SoundSystem::SetSoundFrameUserCallback` function can register a callback function to be called at *timing [1]*.

Internally configure this callback function to set the parameters for the `nn::snd::Voice` instance.

For details, see the `demos/snd/withSdk` demo.

7.3.2 Call nn::snd::WaitForDspSync or nn::snd::SendParameterToDsp from the application

As explained in section 7.2, the application is permitted to call both functions under certain conditions.

Using this method, have the application set the parameters for the `nn::snd::Voice` instance at *timing [1]*.

7.4 nn::snd::SetMasterVolume

In the nw::snd Library, the master volume is controlled by the `nn::snd::SetMasterVolume` function.

The application is prohibited from calling the `nn::snd::SetMasterVolume` function during the period between the call to the `nw::snd::SoundSystem::Initialize` function and the call to the `nw::snd::SoundSystem::Finalize` function.

8 Operation in the System Core

An option was added to NW4C-1.1.0 which enables sound threads started by the nw::snd library to be operated in the system core.

The following sections explain the system core, provide a concrete example of programming to operate sound threads in the system core, and describe limitations that apply to operations in the system core.

8.1 What is the System Core?

The CTR's CPU has two cores: an application core (Core 0) and a system core (Core 1).

Normal applications utilize only the application core, but a feature was added to NW4C-1.1.0 which enables nw::snd sound threads (and only these) to be transferred to the system core for processing.

8.2 How to Operate in the System Core

The sample code below shows how the sound thread can be operated in the system core.

Code 8-1 Operation in the System Core

```
// Initialization of sound system
{
    nw::snd::SoundSystem::SoundSystemParam param;
    param.soundThreadCoreNo = 1;            // This line added
    size_t workMemSize = nw::snd::SoundSystem::GetRequiredMemSize( param );
    m_pMemoryForSoundSystem = MemAlloc( workMemSize );

    nw::snd::SoundSystem::Initialize(
           param,
           reinterpret_cast<uptr>( m_pMemoryForSoundSystem ),
           workMemSize );
}
```

The sound system is initialized using `nw::snd::SoundSystem::SoundSystemParam`, but by setting the `soundThreadCoreNo` member to 1 you can make the sound threads that are started inside the `nw::snd` library operate in the system core.

There is no other code that is added or deleted by the user.

8.3 Processes That Can Run in the System Core

The following two kinds of processes can run in the system core:

- nw::snd library sound thread processes
- Effects provided by the CTR_SDK

The CTR_SDK provides two kinds of effects, namely `nn::snd::FxDelay` and `nn::snd::FxReverb`. These effects provided by the CTR_SDK are restricted in that only one effect can be configured for any given path. For details, see the reference manual for the `nw::snd::SoundSystem` class.

The effects provided by NW4C, on the other hand, always operate in the application core. Furthermore, when the sound thread is not operating in the system core, the effects provided by the CTR_SDK will operate in the application core.

Table 8-1 Relation between cores where sound thread is operating and effect is operating

Core in which sound thread is operating	CTR_SDK Effects	NW4C Effects
System core	Operate in system core	Operate in application core
Application core	Operate in application core	Operate in application core

8.4 Limitations when Operating in System Core

- `nw::snd::FxReverb` and `nw::snd::FxDelay` effects as well as effects inherited from `nw::snd::FxBase` and created by the user are all processed in the application core.
- The `tick` value obtained with the `nw::snd::SoundSystem::GetSoundThreadTickCount` function is larger than the value would be if the sound thread were operating in the application core. This is because the value includes the processing time of processes with higher priority operating in the system core.
- CTR_Profiler does not obtain the correct profile result. But by profiling with CTR_Profiler in only a special thread it can obtain the correct profile for that just thread.

8.5 Supplemental Information

- The callback function set by the user with the `nw::snd::SoundSystem::SetSoundFrameUserCallback` function is called from the sound thread by a certain feature that was going to be disabled in NW4C-1.2.0, according to what was stated in this document Version 1.3.0. But that feature has not been disabled and is still available. The callback function set by this function always operates in the application core, regardless of which core the sound thread is running.

- In the previous version, it was stated that there were no plans to implement some means of enabling the user to configure calls to callback functions from sound threads, but that the information about the output waveforms from the DSP (which previously could only be obtained from inside the callback function) would become obtainable from a thread on the application side. However, as just mentioned, the `nw::snd::SoundSystem::SetSoundFrameUserCallback` function will not be disabled as had been implied in the previous version, so you can still obtain information about the output waveforms from the DSP by calling the `nn::snd::GetMixedBusData` function from inside this function. You can also obtain the information using the `nn::snd::OutputCapture` class, which was made available from CTR_SDK-0.14.0.

9 Processing During Sleep

9.1 Restrictions During Sleep

If the system is in sleep mode, access to nn::fs is prohibited from the point ACCEPT is returned to the sleep query callback until the wakeup callback arrives. The nw::snd library accesses nn::fs from sound data load threads created inside the library when loading streamd sound data or loading data to the player heap. During sleep mode, the following functions must be called to explicitly stop or recover.

Code 9-1 Functions Called During Sleep

```
nw::snd::SoundSystem::EnterSleep();
nw::snd::SoundSystem::LeaveSleep();
```

The next section describes how to use these functions.

9.2 nw::snd::SoundSystem::EnterSleep, nw::snd::SoundSystem::LeaveSleep

Typical use of these functions is shown in SampleDemos/demo1 included in CTR SDK.

When entering sleep, this call is made while passing n::applet::REPLY_ACCEPT to nn::applet::ReplySleepQuery, but before this the nw::snd library is put into sleep mode using the nw::snd：：SoundSystem::EnterSleep function.

The wakeup callback in demo1 calls the s_AwakeEvent.Signal function and control is returned from the s_AwakeEvent.Wait function.

In order to recover the nw::snd library from sleep mode later, you must call the nw::snd::SoundSystem::LeaveSleep function.

This represents all of the sleep-related processes required by the nw::snd library.

Code 9-2 Sleep Process Sample Code

```
void SleepHandler::SleepSystem( void )
{
    // Return without doing anything if no instruction arrives.
    if ( !IsSleepRequested() )
    {
        return;
    }

    nw::snd::SoundSystem::EnterSleep();  // Enter sleep mode

    nn::applet::ReplySleepQuery(nn::applet::REPLY_ACCEPT);
```

```
    s_AwakeEvent.Wait();

    nw::snd::SoundSystem::LeaveSleep();  // Recover from sleep mode

    nn::applet::ClearSleepNotificationState();
}

/*---------------------------------------------------------------------------*
    Callback called when recovering from sleep
 *---------------------------------------------------------------------------*/
void SleepHandler::AwakeCallback( uptr arg )
{
    NN_UNUSED_VAR(arg);
    s_AwakeEvent.Signal();
}
```

10 Revision History

Version	Revision Date	Class	Description
1.6.1	2011/01/12	Added	• 9 Processing During Sleep
		Changed	• 4.3.1 The Player Heap Since specifications from NW4C-1.3.0 have changed so that data other than sequence sounds can be loaded in the player heap, the note stating that “only sequence data can be loaded in the player heap” has been deleted, and text stating that “in addition to sequence data, all sound data can be loaded” has been added. • 4.3.2 Using the Player Heap Corrected mistaken function name (from `nw::snd::SoundSystem::InitSoundSystem` to `nw::snd::SoundSystem::Initialize`)。
1.6.0	2010/12/24	Changed	• 7.1.2 Synchronous Processing Changed the description of about performing operations on voiced obtained independently to match changes to specifications made beginning from CTR_SDK-0.14.15 and NW4C-1.2.15. • 7.1.2 Synchronous Processing 7.2 nn::snd::WaitForDspSync, nn::snd::SendParameterToDsp Noted that the caller of the nn::snd::WaitForDspSync and , nn::snd::SendParameterToDsp functions has changed beginning from CTR_SDK-0.13.0.
1.5.1	2010/12/13	Changed	• 7.1.2 Synchronous Processing Revised notes on operating voiced obtained independently, because this is now handled differently under the current implementation.
1.5.0	2010/11/15	Added	• 7.1.1 Precautions When Threads Are Running in the System Core • 7.1.2 Synchronous Processing Deleted Section 8.4, How to Use in Combination with the CTR_SDK nn::snd Library, and information recollected here.
		Deleted	• 8.4 How to Use in Combination with the CTR_SDK nn::snd Library
		Changed	• 8.5 Supplemental Information Corrected a kanji conversion mistake (changed 鳴りました to なりました。)
1.4.0	2010/11/04		• Revised the text of Chapter 8 Operations in the System Core because he behavior when operating in the system core has changed.
1.3.0	2010/10/04		• Added chapter 8 Operations in the System Core
1.2.0	2010/06/28		• Added section 7. Notes About Using the nn:snd Library

Version	Revision Date	Class	Description
1.1.0	2010/03/19		Added chapter 5 Sound Actors. • Added chapter 6 3D Sound. • Revised chapter 3 Quick Start: Revised sample code to meet NW4C-0.5.0 description. Slightly adjusted explanatory text.
1.0.1	2010/03/05		• Revised cover page: Fixed version numbering, from Ver x.y to Ver. x.y.z. Fixed title, from Sound Programmer Guide to Sound Programmer's Guide. • Reformatted code sections throughout.
1.0.0	2010/01/26		• Added, deleted, and revised text throughout to update for NW4C-0.4.1 features. • Revised section 4.2.3 SoundMemoryAllocatable Class to suggest using the `originalSoundHeap` demo as a reference. • Revised section 4.3 Player Heap to explain that only sequence sound data can be loaded into a player heap. Also suggested using the `playerHeap` demo as a reference. • Deleted chapter 5 Sound Actors. • Deleted chapter 6 3D Sound. • Deleted chapter 7 Animation Sounds. • Deleted chapter 8 Cautions When Using the AX Library. • Deleted chapter 9 Notes. • Deleted chapter 10 The `snd` Library Structure.
-	2009/11/11		• Added note(s) stating that this document is based on NintendoWare for Revolution.
-	2009/10/30		• Initial version.