

# NITRO-Composer Sound Programmer Guide

Version 1.2.3

**The contents in this document are highly  
confidential and should be handled accordingly.**

**Confidential**

These coded instructions, statements, and computer programs contain proprietary information of Nintendo of America Inc. and/or Nintendo Company Ltd. and are protected by Federal copyright law. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

# Table of Contents

---

1	Introduction .....	7
2	Sound Program Development Environment .....	8
2.1	NITRO System Build Environment .....	8
2.2	File Organization .....	8
2.2.1	Library File .....	8
2.2.2	Header File.....	8
2.2.3	Sound Data .....	8
2.2.4	ARM7 Component.....	8
3	Basic Implementation Example .....	9
3.1	Development Environment .....	9
3.1.1	Makefile.....	9
3.1.2	ROM storage file .....	10
3.1.3	Operational Procedure .....	10
3.2	The NitroMain Function .....	10
3.3	Basic setup.....	12
3.3.1	Initializing the OS and Other Processes.....	12
3.3.2	Initializing the Sound Library .....	12
3.3.3	Creating the Sound Heap.....	12
3.3.4	Initializing the Sound Archive .....	12
3.3.5	Setting Up the Player .....	13
3.3.6	Stream library initialization.....	13
3.3.7	Sound Frame Processing.....	13
3.4	Loading Sound Data.....	13
3.4.1	Loading Groups.....	14
3.5	Sequence Operation.....	14
3.5.1	Sound Handles.....	14
3.5.1.1	Using Sound Handles.....	14
3.5.1.2	What is a Sound Handle?.....	14
3.5.1.3	Disconnecting the Sequence .....	14
3.5.1.4	Tips for Creating Sound Handles.....	15
3.5.2	Sequence Playback.....	15
3.5.3	Sequence Archive Playback.....	15
3.5.4	Stopping the Sequence .....	15
3.6	Other Demos .....	16
3.6.1	stream .....	16
3.6.2	stream-2 .....	16
3.6.3	stream-3.....	16
3.6.4	moveVolume .....	16

3.6.5	onMemory .....	16
3.6.6	reverb .....	16
3.6.7	effect.....	16
3.6.8	outputEffect .....	17
3.6.9	sampling .....	17
3.6.10	waveout .....	17
3.6.11	micThrough.....	17
3.6.12	driverInfo.....	17
4	Heap Operations .....	18
4.1	Overview.....	18
4.2	Memory Management Basics .....	18
4.2.1	The Sound Heap and the Player Heap.....	18
4.2.2	Appropriate Usage of the Two Heaps.....	18
4.3	Sound Heap Operations .....	18
4.3.1	Clearing the Heap .....	18
4.3.2	Restoring the Heap to the Previous State.....	19
4.3.3	Multiple Sound Heaps .....	19
4.4	Player Heap Operations .....	20
4.4.1	Deleting the Player Heap.....	20
5	Stream Playback .....	21
5.1	Initializing the stream library .....	21
5.1.1	Stream thread.....	21
5.1.2	Stream buffer .....	21
5.2	Stream operations .....	22
5.2.1	The stream handle .....	22
5.2.2	Stream playback.....	22
5.2.3	Stopping a Stream .....	22
5.2.4	Pausing a Stream .....	22
5.3	Avoid Interrupting Streams .....	23
5.3.1	Stream Thread.....	23
5.3.1.1	Disabling Interrupts .....	23
5.3.1.2	DMA.....	23
5.3.1.3	Interrupt Handler Process.....	23
5.3.1.4	Higher Priority Threads.....	23
5.3.2	Accessing the Card / Backup Media.....	23
5.3.3	Stream Buffer .....	23
5.3.4	Simultaneous Playback .....	24
6	Cautions .....	25
6.1	Sound Processes in Sleep Mode.....	25
6.1.1	Playback Status After Being Awakened .....	25

6.1.2	Wait While Stabilizing Operations After Being Awakened .....	25
7	Library Organization .....	26
7.1	Library Organization .....	26
7.2	Player Library .....	26
7.3	Sound Archive Player Library .....	27
7.4	Sound Archive Stream Library .....	28
7.5	Stream Library .....	28
7.6	Sound Archive Library .....	29
7.7	Sound Heap library .....	29
7.8	Capture Library .....	29
7.9	Waveform Playback Library .....	30

## Code

Code 3-1	Makefile .....	9
Code 3-2	The NitroMain Function .....	10
Code 3-3	Initializing the Sound Library .....	12
Code 3-4	Sound Heap Creation .....	12

## Tables

Table 7-1	Player Library Functions .....	<b>Error! Bookmark not defined.</b>
Table 7-2	Sound Archive Player Library Functions .....	<b>Error! Bookmark not defined.</b>
Table 7-3	Sound Archive Stream Library Functions .....	28
Table 7-4	Stream Library Functions .....	<b>Error! Bookmark not defined.</b>
Table 7-5	Sound Archive Library Functions .....	<b>Error! Bookmark not defined.</b>
Table 7-6	Sound Heap Library Functions .....	<b>Error! Bookmark not defined.</b>
Table 7-7	Capture Library Functions .....	<b>Error! Bookmark not defined.</b>
Table 7-8	Waveform Playback Library Functions .....	<b>Error! Bookmark not defined.</b>

## Figures

Figure 4-1	Restoring the Previous State .....	19
Figure 7-1	Library Organization Diagram .....	26

## Revision History

Version	Revision Date	Description
1.2.3	03/28/2005	<ul style="list-style-type: none"> <li>Added a description of the <code>driverInfo</code> demo</li> </ul>
1.2.2	01/31/2005	<ul style="list-style-type: none"> <li>Added a description of the <code>micThrough</code> demo</li> <li>Supplement to the description of the waveform playback library</li> <li>Changed "NITRO" to "Nintendo DS"</li> </ul>
1.2.1	12/06/2004	<ul style="list-style-type: none"> <li>Added the description of stream-2 and stream-3 demos</li> </ul>
1.2.0	10/12/2004	<ul style="list-style-type: none"> <li>Added description of changing to sleep mode</li> <li>Added description to avoid interrupting streams</li> <li>Added description of <code>sampling</code> demo and <code>outputEffect</code> demo</li> </ul>
1.1.2	09/16/2004	<ul style="list-style-type: none"> <li>Unified the name of <code>.sadl</code> files as "sound label files"</li> </ul>
1.1.1	09/02/2004	<ul style="list-style-type: none"> <li>Revised due to the change in the sample source code</li> </ul>
1.1.0	08/10/2004	<ul style="list-style-type: none"> <li>Added a description of stream playback</li> <li>Added a description of stream library</li> </ul>
1.0.0	07/20/2004	<ul style="list-style-type: none"> <li>Revised text to reflect the addition of the Waveform Playback Library</li> <li>Revised text to reflect the addition of effect functionality</li> <li>Changed the file extension from <code>.bin</code> to <code>.srl</code></li> <li>Revised the description of the ARM7 component</li> </ul>
0.4.0	6/1/2004	<ul style="list-style-type: none"> <li>Revised text to reflect file system support</li> <li>Revised text to reflect the ability for a player to play multiple sequences</li> <li>Added description of heap operations</li> <li>Changed the library organization</li> </ul>
0.3.0	04-01-2004	<ul style="list-style-type: none"> <li>A complete organizational change was done.</li> <li>Added a description of the library organization</li> <li>Added an overview of the sample demo</li> </ul>
0.2.0	03-18-2004	<ul style="list-style-type: none"> <li>Fixed the makefile of the SoundPlayer</li> <li>Added cautions for <code>OS_EnableIrqMask()</code></li> <li>Added a tempo change function</li> </ul>
0.1.0	03-01-2004	<ul style="list-style-type: none"> <li>Initial version</li> </ul>

# 1 Introduction

This document provides programmers with fundamental information about developing sound programs on the Nintendo DS (DS).

The setup for the NITRO-Composer environment is explained. This is followed by an example that shows how to implement a sound program. The last section explains the structure of the sound library, and lists the type of provided functions.

For a detailed explanation of each function, refer to the function reference.

## 2 Sound Program Development Environment

### 2.1 NITRO System Build Environment

---

NITRO-Composer is part of the NITRO-System. By setting up the NITRO-System build environment, you will be able to use NITRO-Composer.

Refer to the NITRO-System documentation for details.

### 2.2 File Organization

---

#### 2.2.1 Library File

---

The NITRO-SDK and NITRO-System library files listed below must be linked..

```
libsnd.a  
libnnsnd.a
```

#### 2.2.2 Header File

---

Header files that include definitions for created functions must be placed in an include statement using the following statement format:

```
#include <nnsys/snd.h>
```

By listing an include statement for the sound label file (\*.sdl) created by the sound designer as shown below, the sound data can be specified using the label defined by the sound designer instead of the number.

```
#include "../data/sound_data.sdl"
```

#### 2.2.3 Sound Data

---

All sound data sets are stored in a single sound archive file that has the \*.sdat file extension. Set the configuration to ensure that this sound archive file is stored in ROM. An example of how to store this file in ROM is shown in Chapter 3..

#### 2.2.4 ARM7 Component

---

The ARM7 component is stored in the NitroSDK. The sound functionality must be implemented in the ARM7 component. If the ARM7 component is not specified explicitly, the component that implements the sound functionality will be used.



## 3 Basic Implementation Example

This chapter demonstrates a basic implementation using Nitro-Composer for a project called `simple`. The `simple` project can be found in `$NitroSystem/build/demos/snd/simple`.

### 3.1 Development Environment

---

This section explains how to set up the development environment.

#### 3.1.1 Makefile

---

The following is an example makefile. Some of the comments have been omitted.

**Code 3-1 Makefile**

```
#!/ make -f

#-----

SRCS          =      main.c

TARGET_NEF =      main.nef
TARGET_BIN =      main.srl

MAKEROM_ROMROOT = ../data
MAKEROM_ROMFILES = sound_data.sdat

include      $(NITROSYSTEM_ROOT)/build/buildtools/commondefs

#-----

do-build:   $(TARGETS)

include      $(NITROSYSTEM_ROOT)/build/buildtools/modulerrules

#==== End of Makefile =====
```

The basic elements of a makefile are not discussed in this section. Refer to the NITRO-SDK and NITRO-System manuals for makefile information. Setting the two `MAKEROM_.*` variables is crucial.

### 3.1.2 ROM storage file

---

`MAKEROM_ROMROOT` defines the root directory in ROM. `MAKEROM_ROMFILES` defines the files to be stored in the root directory. In other words, the files that have the path `../data/sound_data.sdat` are stored in ROM using the path directory and filename.

### 3.1.3 Operational Procedure

---

If the makefile in 3.1.1 is used, the program builds in the following sequence.

- The `main.c` file registered in SRCS compiles.
- The compiled file is linked to the library, which creates the ARM9 component `main`.
- The ARM9 component `main` combines with the ARM7 component and sound archive which creates `main.srl`
- The `main.srl` file becomes the executable file.

## 3.2 The NitroMain Function

---

First, examine the `NitroMain` function in `src/main.c`. Some of the comments have been omitted.

### Code 3-2 The NitroMain Function

```
void NitroMain()
{
    OS_Init();
    GX_Init();

    // VBlank settings
    OS_SetIrqFunction(OS_IE_V_BLANK, VBlankIntr);
    (void)OS_EnableIrqMask( OS_IE_V_BLANK );
    (void)OS_EnableIrq();
    (void)GX_VBlankIntr(TRUE);

    FS_Init( MI_DMA_MAX_NUM );

    // Initialize sound
    NNS_SndInit();
    heap = NNS_SndHeapCreate( & sndHeap, sizeof( sndHeap ) );
    NNS_SndArcInit( &arc, "/sound_data.sdat", heap, FALSE );
    (void)NNS_SndArcPlayerSetup( heap );
    NNS_SndArcStrmInit( STREAM_THREAD_PRIO, heap );

    // Load sound data
    (void)NNS_SndArcLoadSeq( SEQ_MARIOKART64_TITLE, heap );
    (void)NNS_SndArcLoadSeqArc( SEQ_SE, heap );
```

```
(void)NNS_SndArcLoadBank( BANK_SE, heap );

// Initialize sound handles
NNS_SndHandleInit( &bgmHandle );
NNS_SndHandleInit( &seHandle );

// dummy pad read
Cont = PAD_Read();

//===== Main Loop
while(1)
{
    u16 ReadData;

    SVC_WaitVBlankIntr();

    ReadData = PAD_Read();
    Trg = (u16)(ReadData & (ReadData ^ Cont));
    Cont = ReadData;

    if ( Trg & PAD_BUTTON_A ) {
// start BGM
        (void)NNS_SndArcPlayerStartSeq(&bgmHandle, SEQ_MARIOKART64_TITLE );
    }
    if ( Trg & PAD_BUTTON_B ) {
// stop BGM
        (void)NNS_SndPlayerStopSeq( &bgmHandle, 1 );
    }

    if ( Trg & PAD_KEY_UP ) {
// start SE
        (void)NNS_SndArcPlayerStartSeqArc( &seHandle, SEQ_SE, SE_COIN );
    }

    //---- framework
    NNS_SndMain();
}
}
```

The key points are described in subsequent sections.

## 3.3 Basic setup

---

This section explains fundamental functions--for example, library initialization.

### 3.3.1 Initializing the OS and Other Processes

---

First initialize the OS and other basic processes.

```
OS_Init();
GX_Init();

// VBlank settings
OS_SetIrqFunction(OS_IE_V_BLANK, VBlankIntr);
(void)OS_EnableIrqMask( OS_IE_V_BLANK );
(void)OS_EnableIrq();
(void)GX_VBlankIntr(TRUE);

FS_Init( MI_DMA_MAX_NUM );
```

### 3.3.2 Initializing the Sound Library

---

The sound library must be initialized before any `NNS_Snd` functions are called.

#### Code 3-3 Initializing the Sound Library

```
NNS_SndInit();
```

### 3.3.3 Creating the Sound Heap

---

Create the heap that is used to store sound data.

#### Code 3-4 Sound Heap Creation

```
heap = NNS_SndHeapCreate( &sndHeap, sizeof( sndHeap ) );
```

The first argument is the starting address in memory that is used for the sound heap. The second argument is the size of the sound heap.

The return value is the heap handle. The heap handle is used to allocate memory from the sound heap.

### 3.3.4 Initializing the Sound Archive

---

Initialize the sound archive. The sound archive structure must be allocated statically.

```
NNS_SndArcInit( &arc, "/sound_data.sdat", heap, FALSE );
```

The first argument is the sound archive structure. The second argument is the path to the sound archive on the ROM file system.

The third argument is the heap needed to allocate memory for initialization of sound archives, and the sound heap handle that was just created is used. Note that if the allocated memory is released, the sound archive will no longer be usable.

The fourth argument is a flag that controls the loading of symbol data in the sound archive. If the argument is set to `True`, symbol data is used for debugging. Set the argument to `FALSE` for standard initialization.

---

### 3.3.5 Setting Up the Player

Set up the player.

```
NNS_SndArcPlayerSetup( heap );
```

The player settings in the sound archive determine the setup.

Because the player setup requires memory, enter the heap handle as an argument.

---

### 3.3.6 Stream library initialization

To do stream playback, the stream library must be initialized.

```
NNS_SndArcStrmInit( STREAM_THREAD_PRIO, heap );
```

For details on streams, see Chapter 5.

---

### 3.3.7 Sound Frame Processing

Perform sound library frame processing. This function should be called once for each frame. The location of the function call is not important.

```
NNS_SndMain();
```

---

## 3.4 Loading Sound Data

Before playing a sound sequence, the sound data must be loaded.

```
(void)NNS_SndArcLoadSeq( SEQ_MARIOKART64_TITLE, heap );  
(void)NNS_SndArcLoadSeqArc( SEQ_SE, heap );  
(void)NNS_SndArcLoadBank( BANK_SE, heap );
```

`NNS_SndArcLoadSeq` loads the data that is required to play the sequence `SEQ_MARIOKART64_TITLE`. This function concurrently loads the bank and waveform data in addition to the sequence data.

`NNS_SndArcLoadSeqArc` loads the SE sequence archive. Because sequence archives are associated with multiple banks, bank and waveform data are not loaded automatically. The following

function, `NNS_SndArcLoadBank`, loads the bank data for SE. With this function, both the bank and waveform data are loaded. Therefore, it is unnecessary to load waveform data separately.

### 3.4.1 Loading Groups

---

Sound data is not normally loaded alone, unlike the example. If the sound designer defines a group, the group can be loaded as shown in the following example.

```
(void)NNS_SndArcLoadGroup( GROUP_STATIC, heap );
```

The group defines which data sets to load. By calling `NNS_SndArcLoadGroup`, all the data sets are loaded at once. By loading groups, the data can be loaded without changing the code in the program.

## 3.5 Sequence Operation

---

### 3.5.1 Sound Handles

---

#### 3.5.1.1 Using Sound Handles

A sound handle is required to work with a sequence:

```
NNSSndHandle bgmHandle;  
NNSSndHandle seHandle;
```

Temporarily allocate a sound handle statically. Before using a sound handle, initialize it with the following function.

```
void NNS_SndHandleInit( NNSSndHandle* handle );
```

#### 3.5.1.2 What is a Sound Handle?

A sound handle is an object that controls the sequence after playback. A sound handle can control one sequence. If a sequence playback is successful, that sequence will be linked to a sound handle. From that point and until that link is disconnected, operations for that sound handle will operate the sequence.

#### 3.5.1.3 Disconnecting the Sequence

Sometimes a sequence is manually or automatically disconnected. A sequence can be automatically disconnected if a second sequence attempts to start when a player can play only one sequence. The played back sequence is forcibly stopped. Under these circumstances, the sound handle is involuntarily disconnected from the sequence and disabled. Even if operations are performed on a disabled sound handle, no processing will occur.

This means that the programmer does not need to check if the sequence they played back is still playing. Even if the same process is executed while the sequence is being played or the sequence is stopped, there will be no problems, such as a separate sequence being operated by error.

#### 3.5.1.4 Tips for Creating Sound Handles

When sound is played back without any pauses, as with one-shot sound effects, the same sound handle can be used to play repeated sounds. All sounds can be played simultaneously as long as the number does not exceed the maximum number of simultaneous sequences that are allowed. Each parameter can be changed separately immediately after the playback occurs.

Because continuous sounds such as background music or engine noises must be stopped, each of these sound effects requires a separate sound handle.

### 3.5.2 Sequence Playback

---

The following function plays back a sequence:

```
BOOL NNS_SndArcPlayerStartSeq( NNSndHandle* handle, int seqNo );
```

`seqNo` is the sequence number, and the sounds are ordered as they appear in the sound archive.

If the function executes, the sequence links to a sound handle that is passed in as an argument. From this point, this sound handle can be used to carry out processes (e.g., stopping the sequence).

If the sound handle is already linked to a sequence, the connection to the original sequence is disconnected and the sound handle connects to the new sequence. Because the sequence that disconnects from the sound handle can no longer be controlled directly, be careful; however, when there is no need to control the sequence in the future— for example, when a one-shot sound effect is played— no special requirements are needed.

```
if ( Trg & PAD_BUTTON_A ) {  
    (void)NNS_SndArcPlayerStartSeq(&bgmHandle, SEQ_MARIOKART64_TITLE );  
}
```

### 3.5.3 Sequence Archive Playback

---

The following function plays back the sequence from the sequence archive:

```
BOOL NNS_SndArcPlayerStartSeqArc(  
    NNSndHandle* handle, int seqArcNo, int index );
```

`seqArcNo` is the sequence archive number, which is the order of the sequences in the sound archive.

`index` is the index number of the sequence in the sequence archive. The rest is the same as the playback of the sequence.

```
if ( Trg & PAD_KEY_UP ) {  
    (void)NNS_SndArcPlayerStartSeqArc( &seHandle, SEQ_SE, SE_COIN );  
}
```

### 3.5.4 Stopping the Sequence

---

The following function stops the sequence:

```
void NNS_SndPlayerStopSeq( NNSndHandle* handle, int fadeFrame );
```

Enter the sound handle passed into `NNSndHandle` when played back to `handle`. `fadeFrame` is the fadeout frame. The volume level gradually decreases over the specified number of frames.

```
if ( Trg & PAD_BUTTON_B ) {  
    (void)NNS_SndPlayerStopSeq( &bgmHandle, 1 );  
}
```

## 3.6 Other Demos

---

No descriptions for the functions that are used with the `simple` demo are described. This section contains an overview of other demos. The demo programs for NITRO-Composer are all stored under `$NitroSystem/build/demos/snd`.

### 3.6.1 stream

---

Plays back streams. Stream playback is explained in Chapter 5.

### 3.6.2 stream-2

---

Combines multiple stream data in real time and plays back. It registers the callback function that performs the combining process by using the `NNS_SndArcStrmStartEx2` function.

### 3.6.3 stream-3

---

Applies effects to the stream and plays back. It registers the callback function that processes effects by using the `NNS_SndArcStrmStartEx2` function.

### 3.6.4 moveVolume

---

Changes the volume of a sequence over a period of time.. This demo uses the `NNS_SndPlayerMoveVolume()` function to change the volume. It includes code for a fade-in playback.

### 3.6.5 onMemory

---

The entire sound archives that are loaded into memory. For that purpose, the sound archives are initialized with `NNS_SndArcInitOnMemory()` function.

### 3.6.6 reverb

---

A reverb demo that uses the capture feature. This demo uses `NNS_SndCaptureStartReverb` and `NNS_SndCaptureStopReverb`.

### 3.6.7 effect

---

An effects demo that uses the sound capture feature. It passes the output through a simple low-pass



filter (moving average).

This demo uses `NNS_SndCaptureStartEffect` and `NNS_SndCaptureStopEffect`.

---

### 3.6.8 outputEffect

Effects demo that uses the sound capture feature. Switches between the surround mode and the headphones mode for output.

It uses the `NNS_SndCaptureStartOutputEffect` and `NNS_SndCaptureChangeOutputEffect` functions.

---

### 3.6.9 sampling

Sampling demo that uses the sound capture feature. Calculates output levels using sampling data for the display.

It uses `NNS_SndCaptureStartSampling` and other functions.

---

### 3.6.10 waveout

Plays waveform data directly instead of using sequence playback. It plays back sounds recorded with a microphone. Uses `NNS_SndWaveOutStart` for the playback of waveform data.

---

### 3.6.11 micThrough

Uses the low-level stream library `NNS_SndStrm`. Plays back real-time input from the microphone and applies effects to output sounds.

This demo uses `NNS_SndStrmSetup` and `NNS_SndStrmStart`.

---

### 3.6.12 driverInfo

Displays on-screen sound driver information.

The sound driver information is updated with `NNS_SndUpdateDriverInfo`, and the player information in the sound driver can be obtained with `NNS_SndPlayerReadDriverPlayerInfo`.

## 4 Heap Operations

### 4.1 Overview

---

In the simple demo, memory management functions are not used. In the simple demo, only the sound heap is created using `NNS_SndheapCreate` during initialization, and data is loaded into the memory.

The heap operations are explained in subsequent sections.

### 4.2 Memory Management Basics

---

For information about the basics of memory management, see the Sound System Manual. The following is a brief explanation about memory management.

#### 4.2.1 The Sound Heap and the Player Heap

---

There are two heaps: the sound and player heap.

The sound heap is a stack-based heap that programmers use for loading and deleting data.

The player heap is used for loading data automatically during the sequence playback. Programmers do not need to work directly with the player heap.

#### 4.2.2 Appropriate Usage of the Two Heaps

---

The sound heap loads relatively large blocks of data at system startup and during scene changes. The player heap loads relatively small blocks of data (e.g., BGM data) during sequence playback.

Even though sound and player heap are generally used in this way everything can be managed in the sound heap to improve the load efficiency.

### 4.3 Sound Heap Operations

---

Because the sound heap is stack-based, memory is allocated from the top to the bottom and is released from the bottom to the top. Memory is allocated automatically from the heap when sound data loads. To delete unwanted sound data, the memory regions must be released. There are two ways to release memory regions:

- Clearing the heap.
- Restoring the heap to the previous state.

#### 4.3.1 Clearing the Heap

---

All of the sound data can be cleared from the heap. This process is very simple, but when the used

function is executed, all playing sounds will stop. If the memory region that is used for the initialization of the sound archive is released, the sound archive will no longer be available.

To clear all sound data from the heap, call the `NNS_SndHeapClear` function.

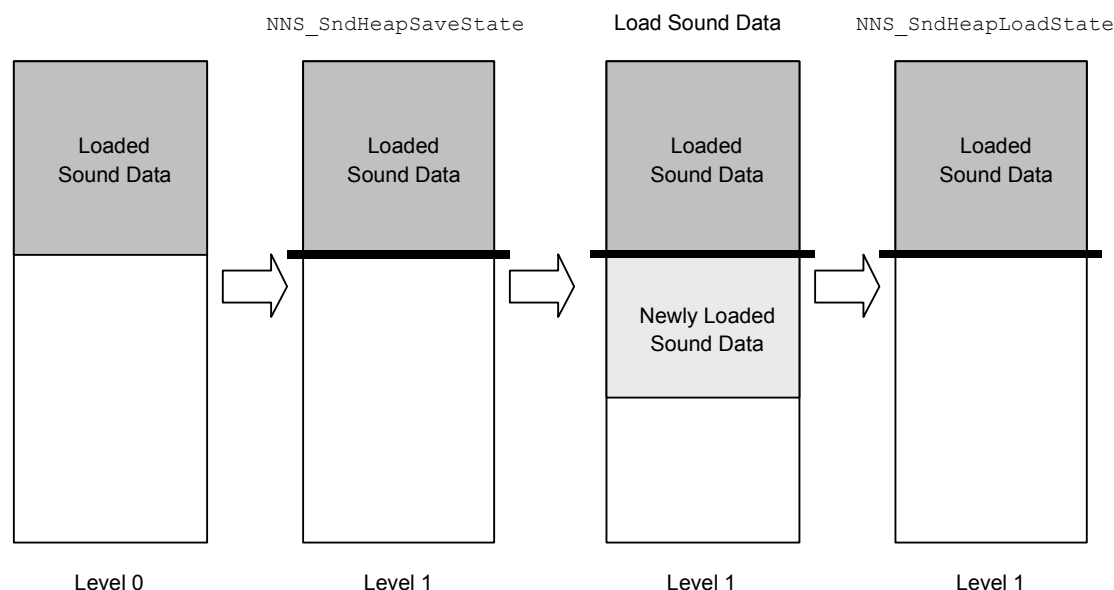
### 4.3.2 Restoring the Heap to the Previous State

Restoring the heap to the previous state is used more frequently than clearing the heap..

`NNS_SndHeapSaveState` saves the current state. After saving, the return value is the hierarchy level of the heap. The hierarchy level indicates the value for the state that was saved. The hierarchy level value can be used to restore the sound heap to the saved state.

After loading several blocks of sound data, calling `NNS_SndHeapLoadState` by passing the hierarchy level value will return the heap to the state that it was in immediately after the call was made to `NNS_SndHeapSaveState`. In other words, all data that was loaded after the call to `NNS_SndHeapSaveState` is deleted.

**Figure 4-1 Restoring the Previous State**



When the data is deleted, already loaded sounds will not be stopped.

Also, `NNS_SndHeapSaveState` can be called repeatedly, and the value of the hierarchy level will increase each time the function is called.

### 4.3.3 Multiple Sound Heaps

Usually a single sound heap is used to restore the heap, but multiple sound heaps can be also used to restore different heap states. If there are several sound heaps, each sound heap can save and restore an individual state.

To use multiple sound heaps, you only need to create multiple heaps using `NNS_SndHeapCreate`. When allocating memory for a sound heap, the heap handle must always be specified. To allocate the memory, specify the sound heap by passing the heap handle return value as the argument of the function.

## 4.4 Player Heap Operations

---

Normally, the programmer does not have to manage the player heap. The player heap is created using the following function.

```
BOOL NNS_SndArcPlayerSetup( NNSSndHeapHandle heap );
```

The sound designer sets the size of the heap needed by the player. The amount of memory is allocated from the sound heap and is passed as an argument in the function to create the player heap.

### 4.4.1 Deleting the Player Heap

---

The player heap allocates memory from the sound heap. When the memory region that is allocated to the player heap is released, the player heap is automatically deleted.

## 5 Stream Playback

This chapter explains stream playback, which was not included in the `simple` demo.

### 5.1 Initializing the stream library

---

To play a stream, the stream library must be initialized.

```
NNS_SndArcStrmInit( STREAM_THREAD_PRIO, heap );
```

The first argument is the stream thread priority. The second argument is the sound heap handle, which is used to allocate the stream buffer.

#### 5.1.1 Stream thread

---

The stream thread is a thread that loads the data from ROM when necessary.

Data is loaded while stream playback is played; the sound stops if the data is not loaded in time. Therefore, the data must be loaded quickly. When a stream thread needs to load data, a stream thread can interrupt other processes running on the main loop of the game and load the data.

If you want the stream thread to interrupt the main processing loop and load data, the stream thread must be assigned a higher thread priority. The main loop (main thread) has a default value of 16. Therefore, specify a value that is less than 16.

#### 5.1.2 Stream buffer

---

To play a stream, a buffer is required to load data. To play a single stream, the buffer needs approximately 2 to 4 KB of memory. This buffer uses part of the sound heap and is passed as the second argument in the function.

Note that once the stream buffer is released, the stream cannot be played.

## 5.2 Stream operations

---

### 5.2.1 The stream handle

---

Just as a sound handle is necessary to play a sequence, a stream handle is required to play a stream.

```
NNSSndStrmHandle strmHandle;
```

The stream handle should be allocated statically for now. Before using the stream handle, initialize a stream function with the following function.

```
void NNS_SndStrmHandleInit( NNSSndStrmHandle* handle );
```

In other respects, a stream handle is identical to a sound handle.

### 5.2.2 Stream playback

---

To play back a stream, call the following function.

```
BOOL NNS_SndArcStrmStart( NNSSndStrmHandle* handle, int strmNo, u32 offset );
```

`strmNo` is the stream number that specifies which stream to play. `offset` specifies in milliseconds when to start playing back in the stream data. Playback generally starts at the beginning of the stream, so set this parameter to zero.

Similar to a when using a sequence, the stream handle is bound to the stream if playback is successful.

### 5.2.3 Stopping a Stream

---

To stop a stream, call the following function.

```
void NNS_SndArcStrmStop( NNSSndStrmHandle* handle, int fadeFrames );
```

`fadeFrames` specifies the number of frames over which the volume should be gradually lowered before the stream stops. If `fadeFrames` is set to a value of zero, the stream stops immediately.

### 5.2.4 Pausing a Stream

---

There is no function to pause a stream. However, a process similar to pausing a stream is possible. The following procedure allows you to create the effect of pausing.

1. When you want to pause the stream, use the following function to obtain the current position of playback.

```
u32 NNS_SndArcStrmGetCurrentPlayingPos( NNSSndStrmHandle* handle );
```

2. Use `NNS_SndArcStrmStop` to stop the stream.
3. Restart the stream playback by passing the playback location as the `offset` argument of `NNS_SndArcStrmStart` and start the stream playback. The playback will start from where the stream stopped.

The stream will not restart at the precise position in the stream.

## 5.3 Avoid Interrupting Streams

---

With stream playback, data is loaded in real-time; therefore, if the sound is not loaded in time, the sound will be interrupted. Here are some tips to avoid interrupting streams.

### 5.3.1 Stream Thread

---

Stream data is loaded with the stream thread. The basic rule is to maintain stream thread processes without delay.

Here are some circumstances that cause delays in the stream thread processes.

#### 5.3.1.1 Disabling Interrupts

Stream threads cannot run while interrupts are disabled. Interrupts should be disabled only for short periods of time.

#### 5.3.1.2 DMA

Stream threads cannot run while DMA is running. By dividing a large DMA into chunks, the delay in stream thread processes can be reduced.

#### 5.3.1.3 Interrupt Handler Process

Stream threads cannot run while an interrupt handler is being processed. Interrupt handler processes should last only for short periods of time.

#### 5.3.1.4 Higher Priority Threads

Stream threads cannot run while higher priority threads are being processed. Higher priority thread processes need to last only for short periods of time or the priority of stream threads needs to be raised.

### 5.3.2 Accessing the Card / Backup Media

---

Stream data cannot be loaded while the card or backup media is accessed. Therefore, divide the bandwidth between the card and stream data.

### 5.3.3 Stream Buffer

---

If the buffer size for stream playback is large, the sound will not be interrupted even if the stream thread process speed is reduced. However, when the buffer size is larger, the process for one stream

thread requires more bandwidth, and a negative effect on the lower priority threads may occur.

#### **5.3.4 Simultaneous Playback**

---

When multiple streams are played simultaneously, the stream thread processes increase and require more bandwidth. The sound may be interrupted even with a short delay in the processes. Exercise caution when playing back multiple streams simultaneously.



## 6 Cautions

### 6.1 Sound Processes in Sleep Mode ---

When changing into sleep mode, sound processes are executed automatically in the library. Programmers only need to call `PM_GoSleepMode`, and the rest of the processes are performed by the library.

However, pay attention to the information in the following sections for Sleep Mode.

#### 6.1.1 Playback Status After Being Awakened ---

The playback status after being awakened from sleep mode may not match the playback status before entering sleep mode. The sound may temporarily become distorted or skip.

When this problem becomes severe, the programmer needs to stop playback before entering sleep mode and restart playback after being awakened.

#### 6.1.2 Wait While Stabilizing Operations After Being Awakened ---

After awakening from sleep mode, the sound circuitry becomes stable after 15 msec. The wait time is processed inside the library function, `PM_GoSleepMode`. Therefore, programmers do not need to pay attention to the wait time.

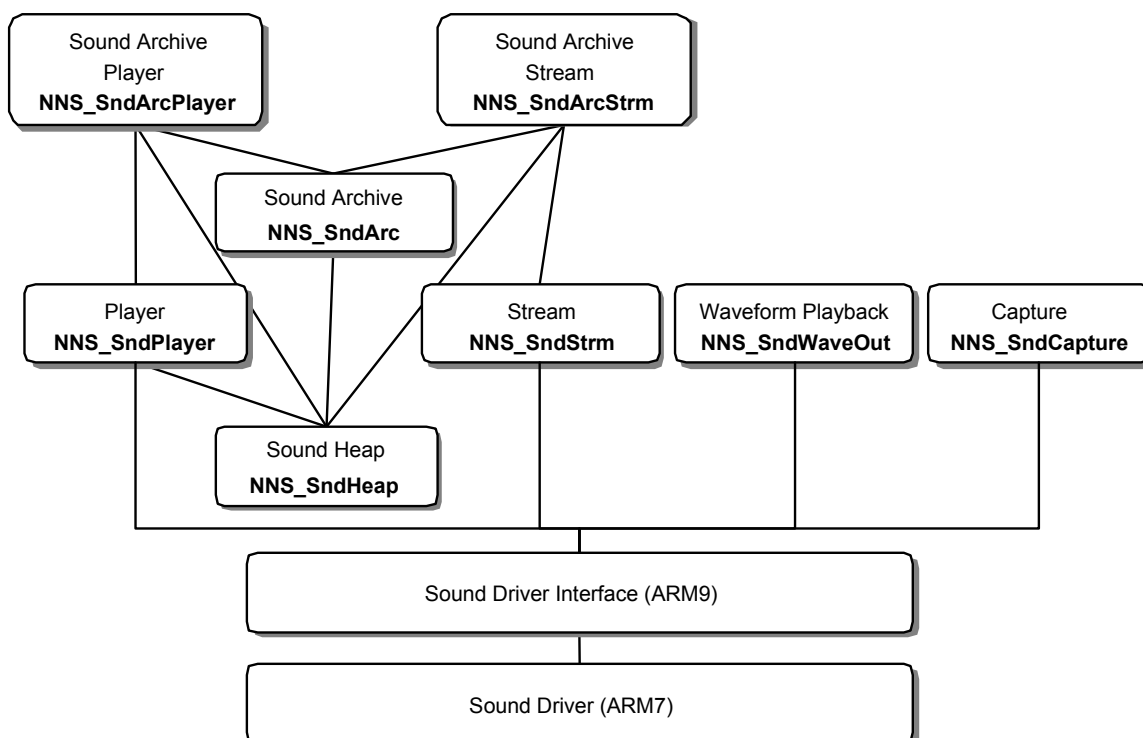
## 7 Library Organization

This chapter describes the organization of the NITRO-Composer library.

### 7.1 Library Organization

The NITRO-Composer library composes several libraries as shown below:

**Figure 7-1 Library Organization Diagram**



The libraries with `NNS_Snd` prefixes are the sound libraries included in `NitroSystem`. Usually, the programmer uses these library functions. The libraries are described in the following sections.

### 7.2 Player Library

The Player Library is the most basic library for playing back sequences. The function prefix set is `NNS_SndPlayer`.

This library is used to change parameters for sequences and stopping sequences. However, the sound archive player functions from the upper library are used for only playback. The sequence playback functions in the player library are used to simply run sequences. For actual sequence playback, complex processing (e.g, data loading) is required. Therefore, sequence playback is processed with the sound archive player.

The player library also includes some functions that operate the sound handles. The function set prefix is `NNS_SndHandle`. The sound handle and player are closely related. Refer to section 3.5.1 Sound Handle for an overview of the sound handle.

The main functions of the Player Library are shown in Table 7-1.

**Table 7-1 Player Library Functions**

Function Name	Description
<code>NNS_SndPlayerStopSeq</code>	Stops the sequence.
<code>NNS_SndPlayerPause</code>	Pauses or restarts the sequence.
<code>NNS_SndPlayerSetTempoRatio</code>	Changes the tempo of the sequence.
<code>NNS_SndPlayerSetVolume</code>	Changes the volume of the sequence.
<code>NNS_SndPlayerSetTrackVolume</code>	Changes the volume of the sequence track.
<code>NNS_SndPlayerSetTrackPitch</code>	Changes the pitch of the sequence track.
<code>NNS_SndPlayerSetTrackPan</code>	Changes the pan (location) of the sequence track.

## 7.3 Sound Archive Player Library

The Sound Archive Player Library plays back the sequence using the sound archive. The function set prefix is `NNS_SndArcPlayer`.

The Sound Archive Player Library is located in the upper-level libraries of the player library and the sound archive library. Sequences can be easily played back using these functionalities.

The main functions of the Sound Archive Player Library are shown in Table 7-2.

**Table 7-2 Sound Archive Player Library Functions**

Function Name	Description
<code>NNS_SndArcPlayerSetup</code>	Sets up the player using the settings in the sound archive.
<code>NNS_SndArcPlayerStartSeq</code>	Plays back the sequence.
<code>NNS_SndArcPlayerStartSeqArc</code>	Plays back the sequence archive.

## 7.4 Sound Archive Stream Library

---

Sound Archive Stream Library is used to play stream data in a sound archive. The function names begin with `NNS_SndArcStrm`.

This library is a upper-level library of the stream library and the sound archive library. The Sound Archive Stream Library uses the functionality of the stream library and the sound archive library to make playing streams simple.

The main functions of the Sound Archive Stream Library are shown in Table 7-3.

**Table 7-3 Sound Archive Stream Library Functions**

Function Name	Description
<code>NNS_SndArcStrmInit</code>	Initializes the sound archive stream library.
<code>NNS_SndArcStrmStart</code>	Plays a stream.
<code>NNS_SndArcStrmStop</code>	Stops a stream.

## 7.5 Stream Library

---

The Stream Library is a low-level library for the playback of streams. The function names begin with `NNS_SndStrm`.

This library is used to play data that is received through communications in real time and for stream playback of waveform data in unique stream data format.

The main functions of the Stream Library are shown in Table 7-4.

**Table 7-4 Stream Library Functions**

Function Name	Description
<code>NNS_SndStrmInit</code>	Initializes a stream.
<code>NNS_SndStrmAllocChannel</code>	Allocates a channel for stream playback.
<code>NNS_SndStrmFreeChannel</code>	Frees a stream playback channel.
<code>NNS_SndStrmSetup</code>	Prepares for stream playback.
<code>NNS_SndStrmStart</code>	Plays a stream.
<code>NNS_SndStrmStop</code>	Stops a stream.

## 7.6 Sound Archive Library

The Sound Archive Library loads sound data in the sound archives and retrieves parameters. The function set prefix is `NNS_SndArc`.

The main functions of the Sound Archive Library are shown in Table 7-5.

**Table 7-5 Sound Archive Library Functions**

Function Name	Description
<code>NNS_SndArcInit</code>	Initializes a sound archive.
<code>NNS_SndArcLoadGroup</code>	Loads sound data in units of the group.

## 7.7 Sound Heap library

The Sound Heap Library manages the sound heap. The function set prefix is `NNS_SndHeap`.

The main functions of the Sound Heap Library are shown in Table 7-6.

**Table 7-6 Sound Heap Library Functions**

Function Name	Description
<code>NNS_SndHeapCreate</code>	Creates the sound heap.
<code>NNS_SndHeapClear</code>	Clears all heap memory.
<code>NNS_SndHeapSaveState</code>	Saves the heap state.
<code>NNS_SndHeapLoadState</code>	Restores the heap state.

## 7.8 Capture Library

Nintendo DS has the sound capture feature. The Capture Library is used for generating effects using the capture feature (e.g., reverb). The function set prefix is `NNS_SndCapture`.

The main functions of the Capture Library are shown in Table 7-7.

**Table 7-7 Capture Library Functions**

Function Names	Description
<code>NNS_SndCaptureStartReverb</code>	Starts the reverb.
<code>NNS_SndCaptureStopReverb</code>	Stops the reverb.
<code>NNS_SndCaptureStartEffect</code>	Starts the effect.
<code>NNS_SndCaptureStopEffect</code>	Stops the effect.

## 7.9 Waveform Playback Library

The Waveform Playback Library provides functionality to play waveform data without using sequence playback. The function set prefix is `NNS_SndWaveOut`.

The Waveform Playback Library is used for playing sampled data captured with a microphone. Use this stream library to play back waveform data that is generated in real time.

The main functions of the Waveform Playback Library are shown in Table 7-8.

**Table 7-8 Waveform Playback Library Functions**

Function Names	Description
<code>NNS_SndWaveOutAllocChannel</code>	Allocates a channel for playing waveform.
<code>NNS_SndWaveOutStart</code>	Starts waveform playback.
<code>NNS_SndWaveOutStop</code>	Stops waveform playback.



© 2005 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed or loaned in whole or in part without the prior approval of Nintendo.