Revolution

AX Sound Pipeline

Version 1.01

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

© 2006-2008 Nintendo
CONFIDENTIAL

RVL-06-0035-001-D
Released: October 17, 2008

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.

Contents

Code Examples

Figures

Tables

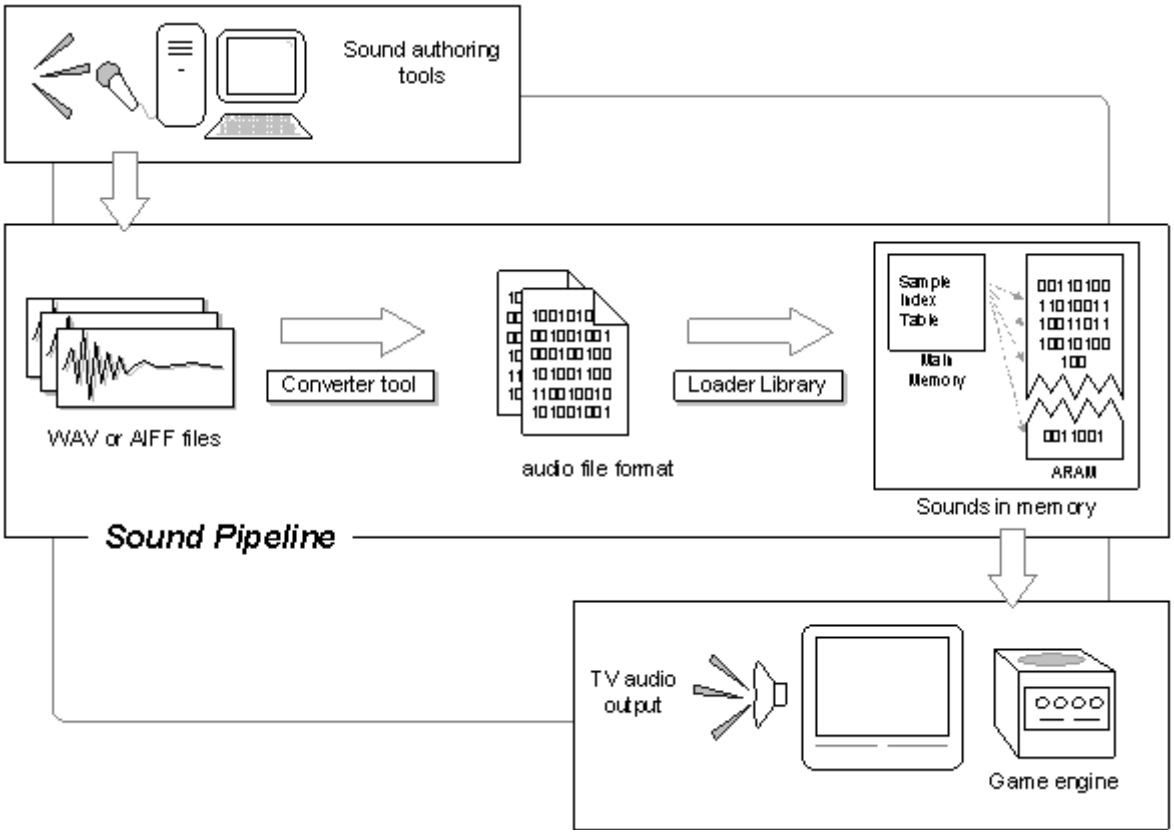
Revision History

Version	Date Revised	Item	Description
1.01	2006/11/21	4.1 4.3.1 4.4.3	Revised figure. Deleted description of the zero buffer. Deleted text related to the zero buffer.
1.00	2006/03/22	-	First release by Nintendo of America Inc.

1 Introduction

The **sound pipeline** refers to the data path between the sound designer and the programmer. It encompasses the process by which sound data is captured for use in a game application. In general, the process is as shown in Figure 1–1.

Figure 1–1 Sound Pipeline Logical Flow



In the process outlined above, a sound designer creates sound effects and saves them as standard WAV or AIFF files. The pipeline imports these files, packs them together and converts them into a format that the loader library can easily read and manipulate during runtime. Once read into memory, the sound effects can be played at will by the game engine.

The Nintendo Revolution SDK provides a prototype pipeline that implements such a process for use with the SDK's native audio library, AX. Full source code is provided for most aspects of the pipeline. Developers are free to examine and modify the source code as they see fit. Note that this pipeline is nearly identical to that of the GameCube.

Note: The AX Sound Pipeline (SP) is only applicable to *sound effects*. It is unrelated to other audio features of the Nintendo Revolution SDK, such as software streaming. The SP also does not explicitly support the conversion and import of general MIDI instruments. Refer to the `DLS1WT` tool in the Nintendo Revolution SDK for details on using DLS-compliant sound data in your game.

1.1 Document Organization

SP's design divides most naturally along the functional lines of an audio development team:

- **Sound Designers** create sound effects for the game. Sound designers are likely also responsible for organizing the sound data and converting the data for use with SP.
- **Tool Programmers** include developers who wish to customize SP for use with a particular game project, or reuse or tweak SP source code to create an entirely new tool path.
- **Game Engine Programmers** are responsible for using the sound effects in a game. They can use SP directly, or modify its runtime libraries to suit the needs of a particular project.

This document is organized around these development roles, with a chapter devoted to each.

Chapter 2 is appropriate for sound designers or anyone responsible for organizing and importing sound data into the AX Sound Pipeline. This chapter discusses:

- Using the `sndconv.exe` converter tool
- The `sndconv.exe` scripting language
- Supported import formats
- Managing the `sndconv.exe` output files

Chapter 3 describes the technical aspects of the SP converter tool and libraries. Specifically, it covers:

- `sndconv.exe` converter tool architecture
- `SOUNDFILE DLL` and API
- `DSPTOOL DLL` and API
- Implementation details of the `sndconv.exe` script parser
- `sndconv.exe` output data formats

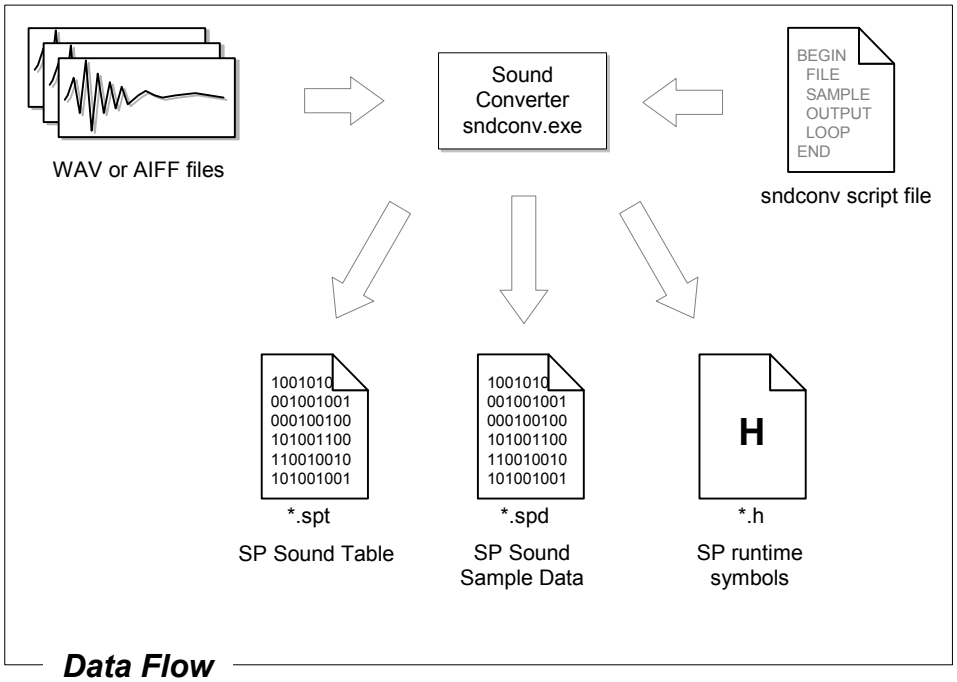
Chapter 4 describes the technical aspects of the SP runtime library, including memory management and playing sounds.

2 Importing Sound Effects into the AX Sound Pipeline

The SP converter tool, `sndconv.exe`, is an x86 command line program that imports a collection of sound files into the SP data format.

A script file specifies each sound file to be converted. The script also defines the associated properties of each sound effect, such as sample rate, size, encoding scheme, and loop data.

Figure 2–1 Data Flow for Importing Sound Effects



The converter tool generates three files:

- **SP Sound Sample Data (`*.spd`):** contains the actual sound samples to be stored in main memory at runtime. The contents of each input WAV or AIFF file are packed together in this file with the appropriate memory alignment.
- **SP Sound Table (`*.spt`):** a table that references and describes the sound samples stored in the *.spd file.
- **SP Runtime Header (`*.h`):** a C header file that contains the symbols with which game engine programmers will access the information in the Sound Table.

For more details on the format and use of these files, see “Data Abstraction and File Formats” on page 32.

The base-name of the output files is derived from the base-name of the script file. For example, if the script file is called “`TEST_SFX.txt`,” then `sndconv.exe` will generate the following output files:

- `TEST_SFX.spd`
- `TEST_SFX.spt`
- `TEST_SFX.h`

2.1 Functionality

2.1.1 Supported Input Formats

The `sndconv.exe` converter tool supports the following input formats:

- Microsoft WAV files
- Macintosh AIFF files (with support for embedded loop points)
- 16-bit and 8-bit PCM sample sizes
- MONO and STEREO input files

2.1.2 Conversion Functions

The `sndconv.exe` tool can perform the following operations on input sound data:

- Compression of 8-bit or 16-bit PCM samples into DSP-ADPCM format
- Conversion of 8-bit or 16-bit PCM samples into 16-bit or 8-bit PCM
- Mixing left and right channels of STEREO data into a single MONO channel
- Extraction of a single channel from a STEREO data file

2.2 Using sndconv.exe

2.2.1 The Command Line

From the command line, you can invoke the `sndconv.exe` program like so:

Code 2–1 sndconv Command Line Syntax

```
bash> sndconv <scriptfile> [-option]

Where:
  <scriptfile>script file (required)

Options are:
  -a    Default output to ADPCM.
  -w    Default output to 16bit PCM.
  -b    Default output to 8bit PCM.
  -h    This help text.
```

The `<scriptfile>` argument in Code 2–1 is required and specifies a text file that contains commands for `sndconv.exe`. These commands specify which sound files to convert and pack. These commands also describe the various attributes of each file, and how they should be converted. For further details, see “Data Abstraction and File Formats” on page 32.

The `-a`, `-w`, and `-b` options specify the *default* output format. The `sndconv.exe` tool uses these defaults if the script file does not specify a desired output format for a given sound file.

If no option is specified, then `sndconv.exe` will use ADPCM as a default output format.

The `sndconv.exe` tool is located in the Nintendo Revolution SDK, under the following path:

```
<SDK install path>/x86/bin
```

Note: `sndconv.exe` requires two DLLs, which are also located under the same path:

- `soundfile.dll`
- `dsptool.dll`

2.2.2 Scripting

The `sndconv.exe` script file specifies which files to convert and how to convert them. An example is shown in Code 2–2:

Code 2–2 Script File Example

```
;
; Text after a semicolon is ignored as commentary.
;

COMMENT
COMMENT This is a COMMENT field. Any text following a COMMENT
COMMENT command will be generated as a comment in the corresponding
COMMENT 'C' header file output. This is useful for annotating the
COMMENT header file from the script
COMMENT

;
; Set source path for sound files
;
PATH C:\sounds

INCLUDE other_script.txt

COMMENT ****************************************
COMMENT Explosion sounds!
COMMENT ****************************************

BEGIN           BIG_EXPLOSION              ; Identifier for sound effect
  FILE          big_exp_pcm16mono.wav      ; Source filename - it's a WAV file!
  SAMPLERATE    22050                      ; Source sample rate, in Hz
  OUTPUT        ADPCM                      ; Will be converted to ADPCM
END

BEGIN           LITTLE_EXPLOSION           ; Identifier for sound effect
  FILE          lil_exp_pcm16stereo.aif    ; Source filename - it's an AIFF file!
  SAMPLERATE    22050                      ; Source sample rate, in Hz
  OUTPUT        ADPCM                      ; Will be converted to ADPCM
END

COMMENT ****************************************
COMMENT Helicopter sounds!
COMMENT ****************************************

BEGIN           CHOPPER                    ; Identifier for sound effect
  FILE          apache_pcm16mono.wav       ; Source filename - it's a WAV file!
  SAMPLERATE    32000                      ; Source sample rate, in Hz
  OUTPUT        16BIT                      ; Output will be in 16bit PCM
  LOOP          117  254                   ; loop point start and end!
END

COMMENT ****************************************
COMMENT Whoosh sound
COMMENT ****************************************

BEGIN           BIG_WHOOSH                 ; Identifier for sound effect
  FILE          bigwhoosh_pcm16stereo.aif  ; Source filename - it's an AIFF file
  SAMPLERATE    32000                      ; Source sample rate, in Hz
  OUTPUT        16BIT                      ; Output will be 16bit also
  LOOP          312 423                    ; loop start and end points
  MIX           COMBINE                    ; Source sample is stereo, downmix to MONO
END
```

```
BEGIN           LEFT_WHOOSH               ; Identifier for sound effect
  FILE          bigwhoosh_pcm16stereo.aif  ; Source filename – it's an AIFF file
  SAMPLERATE    32000                     ; Source sample rate, in Hz
  OUTPUT        16BIT                     ; Output will be 16bit also
  LOOP          312 423                   ; Loop start and end points
  MIX           LEFT                      ; Extract left channel only
END

BEGIN           RIGHT_WHOOSH              ; Identifier for sound effect
  FILE          bigwhoosh_pcm16stereo.aif  ; Source filename – it's an AIFF file
  SAMPLERATE    32000                     ; Source sample rate, in Hz
  OUTPUT        16BIT                     ; Output will be 16bit also
  LOOP          312 423                   ; Loop start and end points
  MIX           RIGHT                     ; Extract left channel only
END

COMMENT ***************************************
COMMENT Beep sound
COMMENT ***************************************

BEGIN           WARNING_BEEP              ; Identifier for sound effect
  FILE          beep_pcm8stereo.aif       ; Source filename – it's an AIFF file
  SAMPLERATE    11025                     ; Source sample rate, in Hz
  OUTPUT        16BIT                     ; Will be converter to 16bit
  MIX           COMBINE                   ; Source sample is stereo, downmix to MONO
END
```

2.2.2.1 Command syntax

Script files have the following basic structure:

Code 2–3 Basic Script Structure and Syntax

```
; Script comments!
;
;


PATH <path specification>

INCLUDE <other script file>

COMMENT <optional comment field>

BEGIN <sound effect name>
  attribute1 <parameter>
  attribute2 <parameter>
  …
END

COMMENT <optional comment field>

BEGIN <another sound effect name>
  attribute1 <parameter>
  attribute2 <parameter>
  …
END
…
```

(1) The INCLUDE Command

Code 2–4 The INCLUDE Command

```
INCLUDE <script file>
```

The INCLUDE command specifies the path of another script file to be included for processing. The path can be relative to the directory established by the last PATH command, if any, or it can be absolute.

This command is optional and can be issued at any point in the script (outside of BEGIN–END clauses).

(2) The PATH Command

Code 2–5 The PATH Command

```
PATH <path specification>
```

The PATH command specifies an absolute or relative path to the directory from which subsequent sound files will be processed. You can issue multiple PATH commands in a script to change directories as needed. PATH commands must exist outside of BEGIN…END clauses. The path must not contain spaces.

This command is optional. If omitted, sndconv.exe will use the directory from which the tool was invoked as the current path.

(3) The Sound Effect Clause

Code 2–6 The Sound Effect Clause

```
BEGIN <sound effect name>
…
END
```

The BEGIN and END commands delimit a clause within which you define the *attributes* of a sound effect.

The field <sound effect name> must be a C-compatible symbol that uniquely identifies the sound effect. When generating the header file, sndconv.exe will collect all sound effect names and automatically enumerate them.

Note: Each sound effect name MUST be unique, otherwise the C header file will fail to compile. Also, BEGIN–END clauses cannot be nested.

(4) The COMMENT Command in C Header Files and Script Files

Code 2–7 The Header File COMMENT Command

```
COMMENT <commentary text for C-header file>
```

The COMMENT command specifies text that must appear as a comment in the C header file. Everything on the line after a COMMENT command will be preserved as text in the C header file (preceded by "//").

Code 2–8 Script Comment

```
; <comment text>
```

Everything after a semicolon (";") is ignored as script commentary.

2.2.2.2 Attributes

This section describes the keywords reserved for defining the attributes of a particular sound effect.

(1) The FILE attribute

Code 2–9 The FILE Attribute

```
FILE <filename>
```

Where <filename> specifies a file (in the current PATH) to be processed. The filename must not contain any spaces.

The sndconv.exe tool will automatically determine the file type by examining the file itself. If the file is neither WAV- nor AIFF-encoded data, the tool will generate an error message and ignore the sound file.

If sndconv.exe cannot find the file, it will issue a warning and continue processing the script.

By default, sndconv.exe will extract the following information from each sound file (depending on type):

Table 2–1 Intrinsic Attributes of Supported Sound File Formats

File Type	Attributes			
	Sample Rate	Bits per sample	Number Channels	Loop Points
WAV	✔	✔	✔	✗
AIFF	✔	✔	✔	✔

Note: WAV files do not support the encoding of loop-point information.

The FILE attribute is **required**.

(2) The SAMPLERATE Attribute

Code 2–10 The SAMPLERATE Attribute

```
SAMPLERATE <source sample rate>
```

Where <source sample rate> is an integer specifying the base sample rate of the sound effect, in Hertz.

This attribute is optional. If omitted, sndconv.exe will use the sample rate encoded in the sound file.

(3) The LOOP Points Attribute

Code 2–11 The LOOP Points Attribute

```
LOOP <loop start>  <loop end>
```

The LOOP attribute specifies the loop start and loop end points of a sample. The <loop start> parameter specifies the first sample played within the loop. The <loop end> parameter specifies the very last sample played within the loop.

Note: For these parameters, samples are counted starting from **zero**. For example, if a loop starts on the 14th sample in the file, then the <loop start> parameter must be set to **13**.

The LOOP attribute is optional. If omitted, loop point information encoded within the sound file (if any) will be used by default. Otherwise, the specified loop points will override the encoded data.

Note: This applies to AIFF files only, as WAV files do not support the encoding of loop point information.

(4) The MIX Attribute

Code 2–12 The MIX attribute

```
MIX <mix operation>
```

The MIX attribute specifies how to handle STEREO sound files. The following operations are supported:

Table 2–2 MIX Operations

MIX Operation	Description
COMBINE	The LEFT and RIGHT channel samples will be mixed together to generate a MONO channel.
LEFT	Only the LEFT channel data will be extracted from the sound file.
RIGHT	Only the RIGHT channel data will be extracted from the sound file.

This attribute is optional. If omitted, the tool will COMBINE stereo files by default.

(5) The OUTPUT Attribute

Code 2–13 The OUTPUT Attribute

```
OUTPUT <conversion operation>
```

The OUTPUT attribute specifies the output format of the sound data. The following conversions are supported:

Table 2–3 OUTPUT Arguments

Conversion Operation	Description
8BIT	Sound data will be stored as 8-bit PCM.
16BIT	Sound data will be stored as 16-bit PCM.
ADPCM	Sound data will be stored as DSP-ADPCM.

This attribute is optional. If omitted, sndconv.exe will use the default output format specified by the -a, -w, or -b command line options. If no command line option is specified, the default output format will be ADPCM.

Note: Sound effects in an SPD file can have different output formats.

2.2.2.3 General Notes on Scripting

- White space is ignored. For example, "BEGIN BLAMMO_32KHZ" is the same as "BEGIN BLAMMO_32KHZ".
- Sound effect names are case-sensitive. Thus, "BEGIN BLAMMO_32KHZ" is unique from "BEGIN Blammo_32KHz".
- Text is parsed line-by-line and must end with a newline. Each line must be less than 255 characters in length.
- Script comments begin with a semicolon and continue to the end of the line.

© 2006-2008 Nintendo

CONFIDENTIAL

3 Tools Programming with the AX Sound Pipeline

3.1 Functionality

The **tool** portion of the AX Sound Pipeline imports standard artist-generated WAV/AIFF files into the SP data format (SPD). Each file contains a single sound effect to be used by the game. Multiple sound effects are packed into a single SP data file to simplify loading and manipulation of the sound data during runtime.

The tool also generates reference information associated with each SPD file. By using the SP runtime library (see Chapter 4), games can easily reference the sound effects packed within the SPD file.

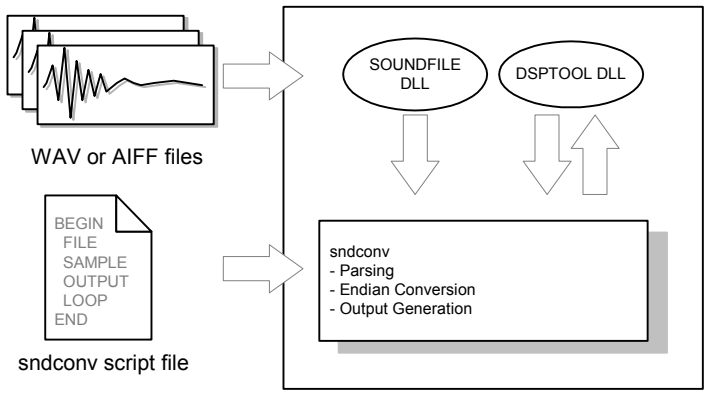
The import process is dictated by a script that the tool parses for filenames, sample attributes, and conversion directives. The import process supports the following conversion directives:

- Compression of 8-bit or 16-bit PCM samples into DSP-ADPCM format
- Conversion of 8-bit or 16-bit PCM samples into 16-bit or 8-bit PCM
- Mixing left and right channels of STEREO data into a single MONO channel
- Extraction of a single channel from a STEREO data file

3.2 Architecture

The Sound Pipeline tool consists of a single executable (`sndconv.exe`) and two WIN32 dynamic-link libraries (`SOUNDFILE.DLL` and `DSPTOOL.DLL`).

Figure 3–1 Basic Architecture and Data Flow for sndconv



The `SOUNDFILE` DLL encapsulates functions for reading and writing WAV and AIFF files. Developers can extend this library to support other formats.

The `DSPTOOL` DLL handles compression of PCM samples into the Nintendo Revolution `DSP-ADPCM` format.

Note: The compression algorithm is proprietary and therefore source code is not provided for this module.

The sndconv program encapsulates the remainder of the tool functions:

- Script parsing
- Sample size conversion (8-bit to 16-bit and vice-versa)
- Mixing and/or extraction of STEREO channels
- Endian conversions
- Generation of output files

3.3 Modules

The various functions of the SP tool are distributed among the following modules:

- The SOUNDFILE DLL
- The DSPTOOL DLL
- The sndconv.exe program

The following sections describe the various functions of the SP tool as they relate to these modules.

3.3.1 The SOUNDFILE DLL

SOUNDFILE is a WIN32 dynamic link library (DLL). This library abstracts the task of reading and writing sound files into a high-level API.

This library currently supports:

- Standard WAV and AIFF file formats
- 8 and 16 bit sample sizes
- Loop markers (AIFF only)

Developers are free to extend this library to support other file formats.

3.3.1.1 Source Code

The source code for the SOUNDFILE dynamic-link library is in the Nintendo Revolution SDK installation directory under the following path:

```
/build/tools/soundfile
```

This library is a Microsoft Visual Studio .NET project that can be accessed by opening the workspace file:

```
/build/tools/soundfile/vc++/soundfile.vcproj
```

Note: In order to invoke the SOUNDFILE DLL from an application, you must include the soundfile.h header file in said application.

3.3.1.2 Data Abstraction

The SOUNDFILE library defines the SOUNDINFO structure as an internal, intermediate descriptor for sound data as it traverses the Sound Pipeline.

Note: The SOUNDINFO structure does not encapsulate the sound data itself. The actual sample data are stored in another buffer, provided by the calling application.

The structure is defined in Code 3–1.

Code 3–1 The SOUNDINFO Structure

```
typedef struct
{

        int     channels;           // Number of channels
        int     bitsPerSample;      // Number of bits per sample
        int     sampleRate;         // Sample rate in Hz
        int     samples;            // Number for samples
        int     loopStart;          // 1 based sample index for loop start
        int     loopEnd;            // 1 based sample count for loop samples
        int     bufferLength;       // buffer length in bytes


} SOUNDINFO;
```

`channels` specifies the number of interleaved sound channels present in the sound data. Monaural data have only 1 channel, while stereo data have 2.

`bitsPerSample` specifies the size of each individual sample. Currently, only 8 or 16 bit sample sizes are supported.

`sampleRate` is the base sampling frequency of the sound data, in Hz.

`samples` specifies the number of samples, **per channel**.

`loopStart` specifies the sample at which a loop, if any, begins.

Note: Samples are counted from 1. If no loop exists in a file, then this value is set to the first sample (1).

`loopEnd` specifies the sample at which a loop, if any, ends.

Note: Samples are counted from 1. If no sample exists in a sound file, then this value is set to the very last sample in the file.

`bufferLength` specifies the length of the buffer required to hold the sample data, in bytes.

3.3.1.3 API

This section describes the functions exported by the `SOUNDFILE` DLL.

(1) The `getSoundInfo` Function

Code 3–2 getSoundInfo()

```
#include "soundfile.h"

int getSoundInfo(char *path, SOUNDINFO *soundinfo)
```

The `getSoundInfo` function opens the file specified by `path` and examines it. If the header is of a valid, supported file type, this function will return the relevant information in the structure pointed to by the `soundinfo` argument.

The `soundinfo` argument must not be NULL.

This function has the following return values:

- SOUND_FILE_SUCCESS
- SOUND_FILE_FORMAT_ERROR
- SOUND_FILE_FOPEN_ERROR

The value SOUND_FILE_FORMAT_ERROR will be returned if the file specified in path is not of a valid, supported sound file type.

(2) The getSoundSamples Function

Code 3–3 getSoundSamples()

```
#include "soundfile.h"

int getSoundSamples(char *path, SOUNDINFO *soundinfo, void *dest);
```

The getSoundSamples function opens the file specified by path and extracts the sound samples therein, as dictated by the information pointed to by SOUNDINFO *soundinfo. The samples will be stored in a buffer pointed to by void *dest.

Before invoking this function, the application must first call getSoundInfo() and retrieve the SOUNDINFO data from the desired file.

The application must allocate memory for the buffer pointed to by void *dest; the size of the buffer can be retrieved from the SOUNDINFO data of the file.

This function has the following return values:

- SOUND_FILE_SUCCESS
- SOUND_FILE_FORMAT_ERROR
- SOUND_FILE_FOPEN_ERROR

The value SOUND_FILE_FORMAT_ERROR will be returned if the file specified in path is not of a valid, supported sound file type.

(3) The writeAiffFile Function

Code 3–4 writeAiffFile()

```
#include "soundinfo.h"

int writeAiffFile(char *path, SOUNDINFO *soundinfo, void *samples);
```

The writeAiffFile function creates an AIFF file as specified by path. The samples argument points to a buffer containing the sound data to be written; the soundinfo structure describes the sound data.

The buffer pointed to by samples must have length equal to or less than soundinfo.bufferLength bytes.

The samples must be little-endian and have 16-bit alignment.

This function has the following return values:

- SOUND_FILE_SUCCESS
- SOUND_FILE_FOPEN_ERROR

(4) The `writeWaveFile` Function

Code 3–5 writeWaveFile()

```
#include "soundinfo.h"

int writeWaveFile(char *path, SOUNDINFO *soundinfo, void *samples);
```

The `writeWaveFile` function creates a WAV file as specified by `path`. The `samples` argument points to a buffer containing the sound data to be written; the `soundinfo` structure describes the sound data.

The buffer pointed to by `samples` must have length no less than `soundinfo.bufferLength` bytes.

The samples must be little-endian and have 16-bit alignment.

Note: Loop information in the `soundinfo` structure will be ignored, as the WAV format does not support loop markers.

This function has the following return values:

- `SOUND_FILE_SUCCESS`
- `SOUND_FILE_FOPEN_ERROR`

3.3.1.4 Using SOUNDFILE

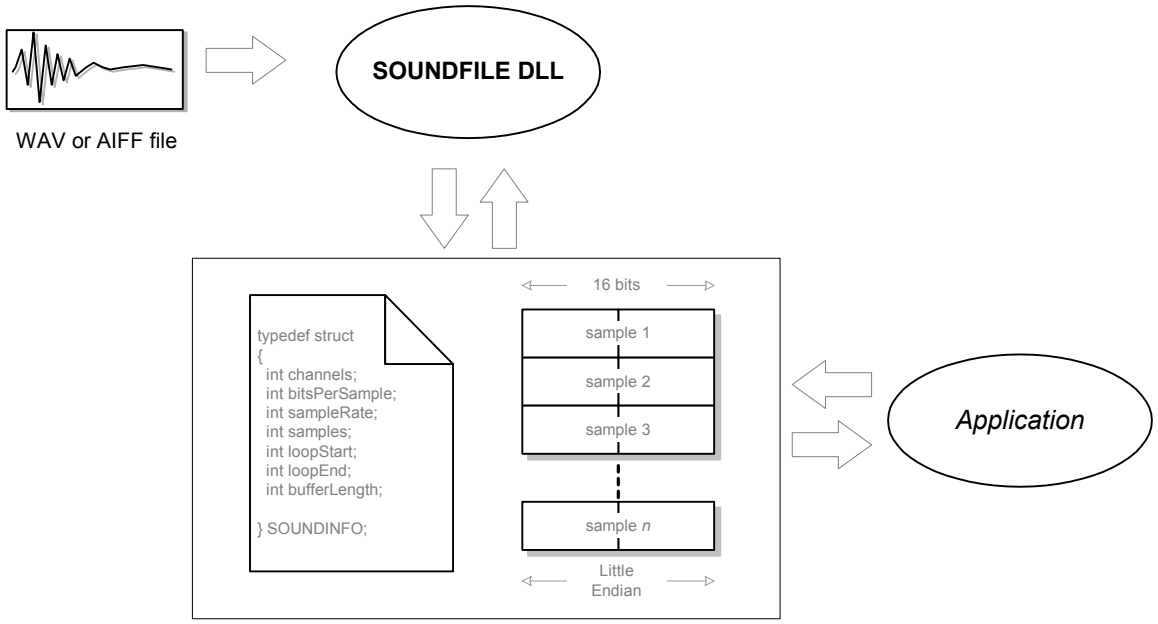
The following code sample illustrates how to load the SOUNDFILE DLL from an application:

Code 3–6 Loading a Dynamic Link Library

```
#include "soundfile.h"

typedef int     (*FUNCTION1)(char *path, SOUNDINFO *info);
typedef int     (*FUNCTION2)(char *path, SOUNDINFO *info, void *dest);
typedef int     (*FUNCTION3)(char *path, SOUNDINFO *info, void *samples);

void main(void)
{
    HINSTANCE   hDLL;
    FUNCTION1   getSoundInfo;
    FUNCTION2   getSoundSamples;
    FUNCTION3   writeWaveFile, writeAiffFile;

    // load up DLL
    if (hDLL = LoadLibrary("soundfile.dll"))
    {
        if (!(getSoundInfo = (FUNCTION1)GetProcAddress(hDLL, "getSoundInfo")))
        {
            printf("GetProcAddress error\n");
            return;
        }

        if (!(getSoundSamples = (FUNCTION2)GetProcAddress(hDLL, "getSoundSamples")))
        {
            printf("GetProcAddress error\n");
            return;
        }

        if (!(writeWaveFile = (FUNCTION3)GetProcAddress(hDLL, "writeWaveFile")))
        {
            printf("GetProcAddress error\n");
            return;
        }

        if (!(writeAiffFile = (FUNCTION3)GetProcAddress(hDLL, "writeAiffFile")))
        {
            printf("GetProcAddress error\n");
            return;
        }
    }
    else
    {
        printf("Cannot load soundfile.dll\n");
        return;
    }

    // do stuff here

    // free the library once you are done
    if (hDLL)
        FreeLibrary(hDLL);
}
```

In general, the SOUNDFILE DLL provides an API and abstraction layer for reading and writing sound data to and from arbitrary data formats.

Figure 3–2 Basic Data Flow of SOUNDFILE DLL



Some notes about the SOUNDFILE data abstraction layer:

- Samples are always stored as little endian words with 16-bit alignment. This facilitates computation on samples (for compression, mixing, or other conversions) by x86 platforms.
- Samples are counted starting from zero.
- Samples are counted per channel. For example: 160 samples of 16-bit monaural sound data is 320 bytes (160 samples x 2 bytes/sample). However, 160 samples of 16-bit *stereo* data is 640 bytes (160 samples x 2 bytes/sample x 2 channels).
- When specifying loop points, the loop start is the first sample played within the loop. Likewise, the loop end sample is the last sample played within the loop.

3.3.2 The DSPTOOL DLL

DSPTOOL is a WIN32 dynamic linked library (DLL). It provides an API for encoding and decoding 16-bit PCM samples to and from the DSP-ADPCM compression format.

The DSP-ADPCM sample format provides (approximately) 3.5:1 compression and is proprietary to the Nintendo Revolution audio DSP. The audio DSP contains special hardware to decompress DSP-ADPCM samples for free.

Note: Source code is not provided for this library.

3.3.2.1 Data Abstraction

The DSPTOOL library defines the ADPCMINFO structure to describe sound data encoded in the DSP-ADPCM format. The structure is given in Code 3–7.

Code 3–7 The ADPCMINFO Structure

```
typedef struct
{

// initial state
        s16 coef[16];
        u16 gain;
        u16 pred_scale;
        s16 yn1;
        s16 yn2;

// loop context
        u16 loop_pred_scale;
        s16 loop_yn1;
        s16 loop_yn2;

} ADPCMINFO;
```

Notes:

- Some members of the structure are always set to zero by the DSPTOOL encoder. These members are included to emphasize the fact that the corresponding registers of the DSP's decoder hardware must be cleared before processing an ADPCM voice.
- The loop context parameters will always be zero after calling encode(). To set these values, you must call getLoopContext() if the sample is indeed looped.
- Sample addresses are not included in the ADPCMINFO structure, because the DSPTOOL library cannot predict where in main memory an ADPCM sound effect will be placed. The calling application is responsible for calculating address offsets for the loop points of a sound effect and passing that information on to the runtime game engine.

3.3.2.2 API

This section describes the functions exported by the DSPTOOL library.

(1) The `getBytesForAdpcmBuffer` Function

Code 3–8 getBytesForAdpcmBuffer()

```
#include "dsptool.h"

U32 getBytesForAdpcmBuffer(u32 samples)
```

The getBytesForAdpcmBuffer function calculates the number of bytes needed to store a sample once it has been ADPCM-encoded, rounded up to the next frame. The samples argument specifies the number of 16-bit PCM samples that are to be encoded.

Note: The number of bytes returned will be rounded up to the next multiple of 8, which is the size of an ADPCM frame. The actual number of bytes needed may therefore be less than the value returned by this function (see getBytesForAdpcmSamples()). This is done because ADPCM sounds must be 8-byte aligned when stored in memory. This function is useful when several sounds must be encoded and packed together.

(2) The `getBytesForAdpcmSamples` Function

Code 3–9 getBytesForAdpcmSamples()

```
#include "dsptool.h"

u32 getBytesForAdpcmSamples(u32 samples)
```

This function calculates the actual number of bytes needed to store a sample once it has been ADPCM-encoded. The `samples` argument specifies the number of 16-bit PCM samples that are to be encoded.

(3) The `getBytesForPcmBuffer` Function

Code 3–10 getBytesForPcmBuffer()

```
#include "dsptool.h"

u32 getBytesForPcmBuffer(u32 samples)
```

The `getBytesForPcmBuffer` function calculates the number of bytes needed to store a sample once it has been decoded from the ADPCM format. This function is useful for determining the amount of memory needed to store data before it has been decoded from ADPCM.

The argument `samples` specifies the number of ADPCM samples that will be decoded.

(4) The `getBytesForPcmSamples` Function

Code 3–11 getBytesForPcmSamples()

```
#include "dsptool.h"

u32 getBytesForPcmSamples(u32 samples)
```

For the given number of `samples`, the `getBytesForPcmSamples` function returns the number of bytes needed to store said samples as 16-bit PCM data. This function is included for symmetry in the API.

(5) The `getSampleForAdpcmNibble` Function

Code 3–12 getSampleForAdpcmNibble()

```
#include "dsptool.h"

u32 getSampleForAdpcmNibble(u32 nibble)
```

For a given `nibble` address, the `getSampleForAdpcmNibble` function will return the corresponding sample in the sound data.

This function assumes that the given nibble address actually points to a valid sample, and does NOT point to a frame header.

Notes:

- ADPCM frames are 16 nibbles in length, with the first two nibbles containing header data. Furthermore, ADPCM frames must start on multiples of 16 nibbles (8 bytes) in memory.
- The sample index returned is counted from zero. In other words, the first sample in an ADPCM sound effect has offset zero.

(6) The `getBytesForAdpcmInfo` Function

Code 3–13 getBytesForAdpcmInfo()

```
#include "dsptool.h"

u32 getBtytesForAdpcmInfo(void)
```

The `getBytesForAdpcmInfo` function returns `sizeof(ADPCMINFO)`.

(7) The `getNibblesForNSamples` Function

Code 3–14 getNibblesForNSamples()

```
#include "dsptool.h"

u32 getNibblesForNSamples(u32 samples)
```

For a given number of `samples`, the `getNibblesForNSamples` function returns the total number of nibbles needed to store them as `ADPCM`-encoded data.

Note: This calculation accounts for frame header overhead.

This function is useful for calculating the nibble-address offset of a particular sample in an `ADPCM` sound effect.

(8) The `getLoopContext` Function

Code 3–15 getLoopContext()

```
#include "dsptool.h"

void getLoopContext(u8 *src, ADPCMINFO *cxt, u32 samples);
```

For a given sound effect that has been `ADPCM`-encoded, the `getLoopContext` function will determine the loop context at the specified loop starting point.

The ADPCM-encoded sound data is pointed to by `src`.

The loop context information will be placed in the `ADPCMINFO` structure pointed to by `cxt`.

The loop start point is specified by the sample offset given in `samples`. For example, if the loop begins at the very first sample of the buffer, then `samples` must be zero. If the loop begins at the 100th sample of the buffer, the offset is 99.

This function has no return value.

(9) The `encode` Function

Code 3–16 encode()

```
#include "dsptool.h"

void encode(s16 *src, u8 *dst, ADPCMINFO *cxt, u32 samples);
```

The `encode` function converts a 16-bit PCM sound effect into the DSP-ADPCM format. The argument `src` points to the 'raw' PCM data. The argument `cxt` points to a buffer into which the `ADPCM` data will be stored. The associated `ADPCM` coefficients and decoder state will be stored in `*cxt`. The argument `samples` specifies the number of samples to convert.

Note: The **raw** PCM data must be in 16-bit little-endian format.

(10)The `decode` Function

Code 3–17 decode()

```
#include "dsptool.h"

void decode(u8 *src, u16 *dst, ADPCMINFO *cxt, u32 samples);
```

The `decode` function decodes `ADPCM` data into raw, 16-bit PCM samples in little-endian format.

The argument `src` points to the `ADPCM` encoded data.

`dst` points to the buffer where the raw PCM samples will be stored. The calling application must allocate this memory for this buffer beforehand.

`cxt` points to the `ADPCM` coefficients and state data that correspond to the data being decoded.

`samples` specifies the number of samples to be decoded.

3.3.2.3 Using DSPTOOL

The following code sample illustrates how to load the DSPTOOL DLL from an application:

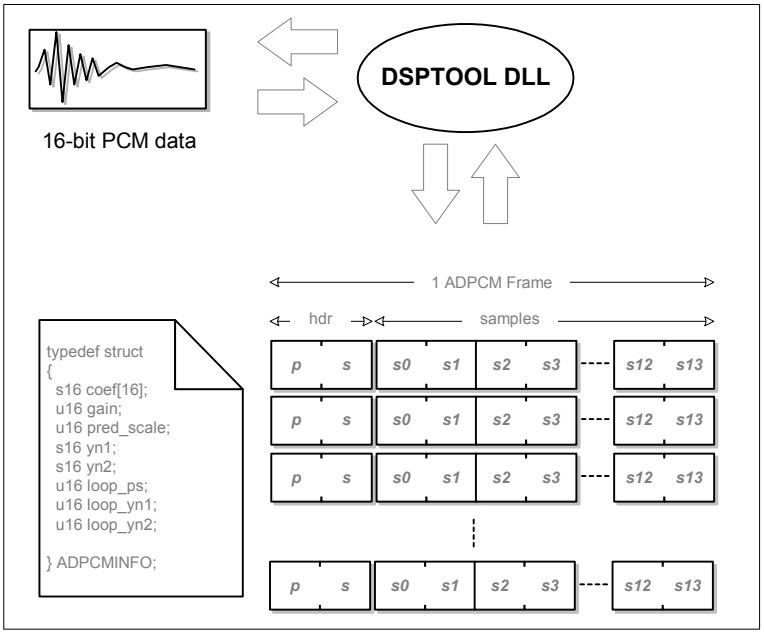
Code 3–18 Loading a Dynamic Link Library

```
#include "dsptool.h"

static HINSTANCE hDll;

typedef u32 (*lpFunc1)(u32);
typedef u32 (*lpFunc2)(void);
typedef void (*lpFunc3)(s16*, u8*, ADPCMINFO*, u32);
typedef void (*lpFunc4)(u8*, s16*, ADPCMINFO*, u32);
typedef void (*lpFunc5)(u8*, ADPCMINFO*, u32);

lpFunc1 getBytesForAdpcmBuffer;
lpFunc1 getBytesForAdpcmSamples;
lpFunc1 getBytesForPcmBuffer
lpfunc1 getBytesForPcmSamples;
lpfunc1 getSampleForAdpcmNibble;
lpfunc1 getNibblesForNSamples;
lpFunc2 getBytesForAdpcmInfo;
lpFunc3 encode;
lpFunc4 decode;
lpFunc5 getLoopContext;


/*---------------------------------------------------------------------------*/
void clean_up(void)
{
    if (hDll)
       FreeLibrary(hDll);
}

/*---------------------------------------------------------------------------*/
int getDll(void)
{
    hDll = LoadLibrary("dspadpcm.dll");

    if (hDll)
    {
        if (!(getBytesForAdpcmBuffer =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getBytesForAdpcmBuffer"
                            ))) return 1;

        if (!(getBytesForAdpcmSamples =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getBytesForAdpcmSamples"
                            ))) return 1;

        if (!(getBytesForPcmBuffer =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getBytesForPcmBuffer"
                            ))) return 1;

        if (!(getBytesForPcmSamples =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getBytesForPcmSamples"
```

```
                            ))) return 1;

        if (!(getNibblesForNSamples =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getNibbleAddress"
                            ))) return 1;

        if (!(getSampleForAdpcmNibble =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getSampleForAdpcmNibble"
                            ))) return 1;

        if (!(getBytesForAdpcmInfo =
                (lpFunc2)GetProcAddress(
                            hDll,
                            "getBytesForAdpcmInfo"
                            ))) return 1;

        if (!(encodeLittleEndian =
                (lpFunc3)GetProcAddress(
                            hDll,
                            "encode"
                            ))) return 1;

        if (!(encodeBigEndian =
                (lpFunc4)GetProcAddress(
                            hDll,
                            "decode"
                            ))) return 1;

        if (!(getLoopContext =
                (lpFunc5)GetProcAddress(
                            hDll,
                            "getLoopContext"
                            ))) return 1;

        return(0);
    }

    return(1);
}

/*---------------------------------------------------------------------------*/
void main (void)
{
        if (getDll)
        {
                clean_up();
                exit(1);
        }

// do stuff here

        clean_up();
}
```

The DSPTOOL library provides services for encoding and decoding sound data, as well as functions for calculating loop contexts, determining ADPCM addresses for loop points, and counting the number of bytes in ADPCM encoded data.

Figure 3–3 Basic Data Flow of the DSPTOOL DLL



Some notes on using the DSPTOOL library and ADPCM samples in general:

- The ADPCM decoding hardware built into the Nintendo Revolution audio system addresses ADPCM samples by nibble (4-bit words). Thus, the DSPTOOL library will calculate loop points as nibble-address offsets. This differs from the SOUNDFILE library, which designates specific samples as loop points.

  Note: SOUNDFILE counts samples starting at zero. Applications must take care to reconcile these two methods when converting looped sound effects between PCM and ADPCM.

- ADPCM data are stored in frames. These frames are 16 nibbles (8 bytes) in length. Header information resides in the first two nibbles.
- ADPCM frames must be 8-byte aligned in main memory.
- Loop start and end addresses must point to actual samples, never to header nibbles. Otherwise, the DSP may behave unpredictably (such as looping unexpectedly when mixing a sound effect, or generating pop/click artifacts).

  Note: DSPTOOL ensures that loop addresses are valid and do not point to header nibbles.

3.3.3 The sndconv Program

The sndconv.exe tool is a WIN32 command-line application. It uses the SOUNDFILE and DSPTOOL libraries to import and compress sound data from standard WAV and AIFF files.

The tool provides a scripting interface so that users can specify a multitude of sound files to import. The program packs these sound files together into a form that an application can easily manipulate during runtime.

The tool generates three output files:

- A C header file that enumerates all of the imported sound effects
- An *.SPT table file that contains parameter information for each sound effect
- An *.SPD file that contains the actual sound samples to be stored in a main memory buffer

3.3.3.1 Source Code

The source code for the `sndconv.exe` application is in the Nintendo Revolution SDK installation directory under the following path:

```
/build/tools/sndconv
```

This program is a Microsoft Visual Studio .NET project and can be accessed by opening the workspace file:

```
/build/tools/sndconv/vc++/sndconv.vcproj
```

3.3.3.2 Program Flow and Implementation Notes

In general, the tool operates like this:

1. Upon execution, the tool parses any command line arguments and searches for the specified script file.
2. The tool maintains a pair of data structures (`SNDCONVDATA` and `ADPCMINFO`) that describe the sound file it is currently processing. The tool clears the data structures upon encountering a `BEGIN` command.
3. The tool updates the various parameters of `SNDCONVDATA` as it reads attributes within the `BEGIN-END` clause.
4. Upon encountering an `END` command, the tool executes the specified conversion operations (if any) and writes the converted sound data to the SPD file. The contents of `SNDCONVDATA` are added to a table, which will be used to generate the SPT file when script processing is complete.
5. If a sound file is converted to `ADPCM`, then the file's associated `ADPCMINFO` data are stored in another table. This table will be appended to the SPT file when script processing is complete.
6. The tool continues to process the script, repeating steps 2-5 until the end of the script is reached.
7. Upon reaching the end of the script, the accumulated entries for `SNDCONVDATA` and `ADPCMINFO` are written to the SPT file.
8. The tool generates the C header file based on the accumulated `SNDCONVDATA` information.

Other notes:

- The tool allows a maximum of 65,536 sound files to be processed and packed in a single SPD file
- Sound effects are enumerated starting from 0x0000

3.3.3.3 Data Abstraction and File Formats

Each sound effect processed by the `sndconv` script is described by an entry in the SPT table. Each entry is a data structure of type `SNDCONVDATA`, defined in Code 3–19:

Code 3–19 The SNDCONVDATA Structure

```
typedef struct
{
    u32 type;

#define SP_TYPE_ADPCM_ONESHOT   0
#define SP_TYPE_ADPCM_LOOPED    1
#define SP_TYPE_PCM16_ONESHOT   2
#define SP_TYPE_PCM16_LOOPED    3
#define SP_TYPE_PCM8_ONESHOT    4
#define SP_TYPE_PCM8_LOOPED     5

    u32 sampleRate;
    u32 loopAddr;
    u32 loopEndAddr;
    u32 endAddr;
    u32 currentAddr;
    u32 adpcm;

} SNDCONVDATA;
```

The `type` parameter specifies the bit size of each sample in the sound effect and whether or not the sound effect is looped.

The `sampleRate` parameter specifies the base sampling frequency, in Hertz.

The `loopAddr` parameter specifies the address at which the loop, if any, begins.

Note: The address specifies the first sample played within the loop.

The `loopEndAddr` parameter specifies the address at which the loop, if any, ends.

Note: The address specifies the last sample played within the loop.

The `endAddr` parameter specifies the address of the last sample in the sound effect.

The `currentAddr` parameter specifies the address of the first sample in the sound effect.

The `adpcm` parameter is an index into the `ADPCMINFO` table that is appended to the SPT file. If the sound effect is an `ADPCM`-encoded sample, then the index points to the relevant entry in the `ADPCMINFO` table.

Note: The addresses in this data structure are offsets into the SPD file. Furthermore, the addressing mode of each address will vary depending on the sample type. For more details, see "3.3.3.4.1 Sample Addressing" on page 35.

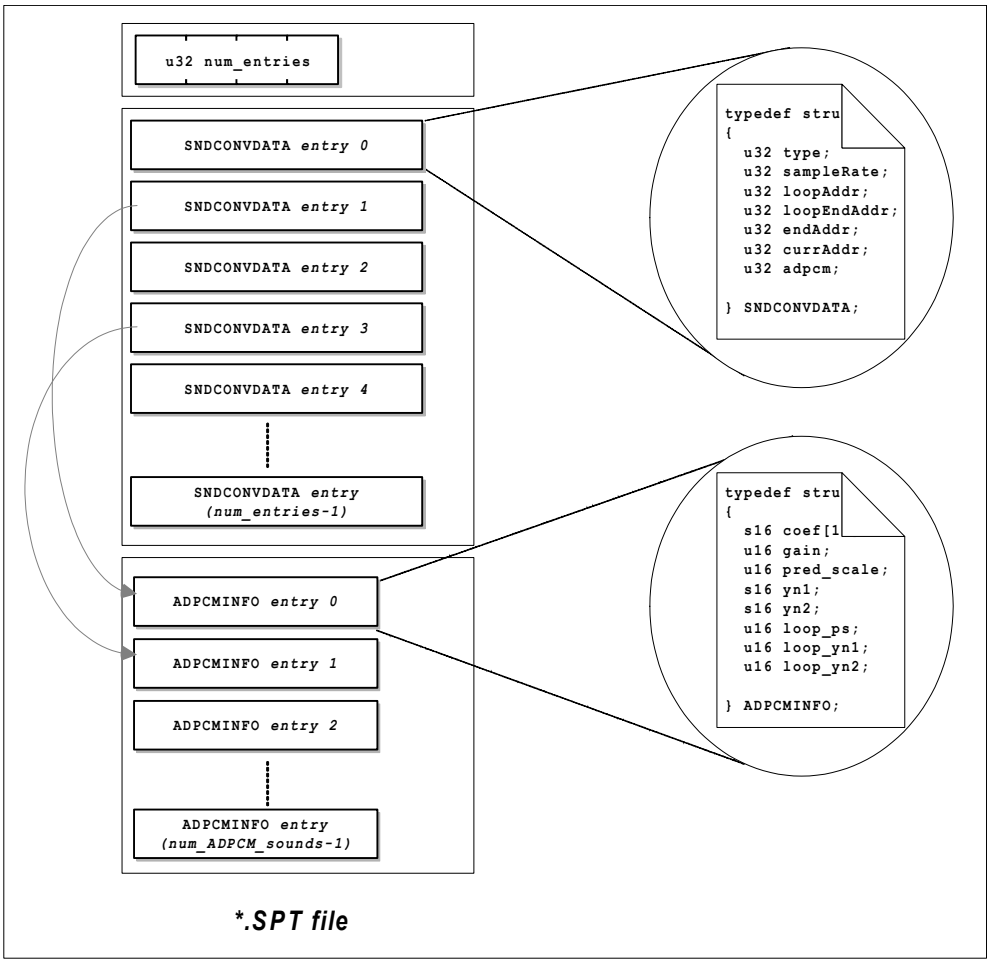
Code 3–20 ADPCMINFO Data Structure

```
typedef struct
{
    // initial state
    u16 coef[16];
    u16 gain;
    u16 pred_scale;
    u16 yn1;
    u16 yn2;

    // loop context
    u16 loop_pred_scale;
    u16 loop_yn1;
    u16 loop_yn2;

} ADPCMINFO;
```

Each ADPCMINFO entry describes the decoding parameters for a given ADPCM-encoded sound effect. The table of ADPCMINFO data is appended to the end of the SPT file.

The format of the SPT file is illustrated below.

Figure 3–4 Format and Internal References of SPT Files



The SPT file is prefaced by an integer (32-bit, unsigned, big-endian), which indicates the number of SNDCONVDATA entries that are present in the file.

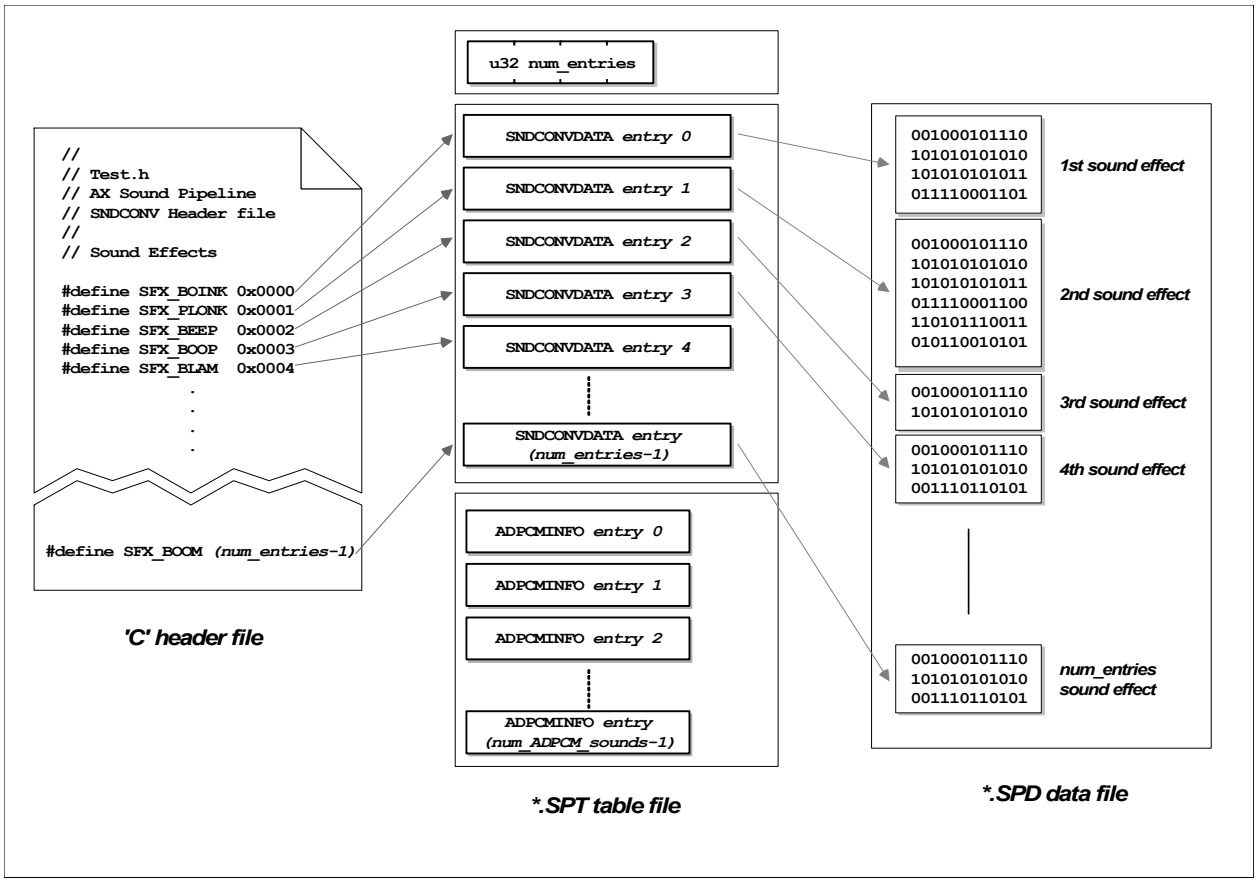
Immediately following the SNDCONVDATA table is another table, which captures the ADPCMINFO data for all sound effects that have been ADPCM encoded.

Note: The number of ADPCMINFO entries is unspecified; however, the remainder of the file contains only the ADPCMINFO table.

In the illustration above, sound effect entries 1 and 3 are ADPCM encoded, and therefore point to the first and second ADPCMINFO entries, respectively.

Each entry in the SNDCONVDATA table corresponds to a sound effect packed into the associated SPD file. Furthermore, a macro definition for each sound effect is placed in the C header file generated by sndconv. The mapping between the sndconv output files is shown in Figure 3–5:

Figure 3–5 Mapping between Header File, SPT File and SPD File



Some notes on the data format of the output files:

- All data in the SPT table file and SPD data file are big-endian
- The sndconv tool packs ADPCM-encoded sound effects on 8-byte boundaries in the SPD data file
- The sndconv tool packs 16-bit sound effects on 2-byte boundaries in the SPD data file
- The sndconv tool packs 8-bit sound effects on byte boundaries in the SPD data file

3.3.3.4 Sample Addressing, Alignment and Loop Point Specification

Perhaps the most vexing aspects of managing sound effects are:

- Sample addressing
- Sample alignment in memory
- Specification of loop points

3.3.3.4.1 Sample Addressing

The DSP contains special hardware to automatically read and, if necessary, decode samples from main memory. The addressing mode of this hardware as it reads from main memory depends on the type of sample being read, as noted below.

Table 3–1 Sample Schemes and Addressing Modes

Sample Encoding Scheme	Addressing Mode
8BIT PCM	Byte
16BIT PCM	Word (16-bit)
`ADPCM`	Nibble (4-bit)

Associated with each sound effect are several addresses that describe its beginning, end, and loop points.

Table 3–2 Associated Addresses

Address	Description
`loopAddr`	Address of first sample in loop. If the sound effect is not looped, this is zero.
`loopEndAddr`	Address of last sample in loop. If the sound effect is not looped, this is zero.
`currAddr`	Address of first sample of sound effect.
`endAddr`	Address of last sample of sound effect.

Each address will access either a byte, word, or nibble, depending on the encoding scheme of the samples.

When generating the SPT table entry for a sound effect, `sndconv` will reconcile the sample type and addressing mode and write the appropriate values into the address parameters.

Note: These addresses are actually *offsets* into the SPD file. This is because the `sndconv` tool cannot predict where in memory the sound effects data will be placed. The Sound Pipeline runtime library (SP) must therefore update each address value by adding the absolute base address at which the SPD data are loaded. The runtime library will reconcile the absolute base address with the addressing mode of each sound effect.

3.3.3.4.2 Sample Alignment in Memory

Another hazard to consider is the alignment of sound effects in main memory as noted in Table 3–3:

Table 3–3 Sample Types and Memory Alignment

Sample Encoding Scheme	Required Main Memory Alignment
8-bit PCM	Sound effect must start on 8-bit boundary.
16-bit PCM	Sound effect must start on 16-bit boundary.
`ADPCM`	Sound effect must start on 64-bit boundary.

Note: Data must be transferred into main memory at 32-byte boundaries. Based on this restriction, `sndconv` automatically pads sound effects to preserve the required alignments.

3.3.3.4.3 Loop Point Specification

Yet another hazard to consider is the specification of loop points.

Loop markers/points are most commonly specified in *samples*. In other words, if a sound effect has a loop-start marker at *n*, then the n^{th} sample (counting from zero) is the first sample played within the loop. The loop-end marker is indicated similarly: if the end marker is at *m*, then the m^{th} sample (counting from zero) is the last sample played within the loop. This is the convention used by many commercial sound editing applications.

Note, however, that AIFF files encode loops differently. Loop start markers are stored as the first sample in a loop, counting from zero. But the end marker is the sample AFTER the last sample played in the loop.

The DSP also accesses loop points differently: its decoding hardware expects loop points to be specified as addresses.

The AX Sound Pipeline reconciles these differences when importing sound effects. Note the calculations carefully when modifying SP or creating your own tool path.

Incorrect loop points may cause discontinuity artifacts in the sound output (such as ‘clicks’ or ‘pops’). Furthermore, if the loop marker of an `ADPCM` sound effect points to a frame header, then the DSP may behave unpredictably (looping sound effects may never end, or sound effects may never start).

4 Game Engine Programming with the AX Sound Pipeline

4.1 Overview

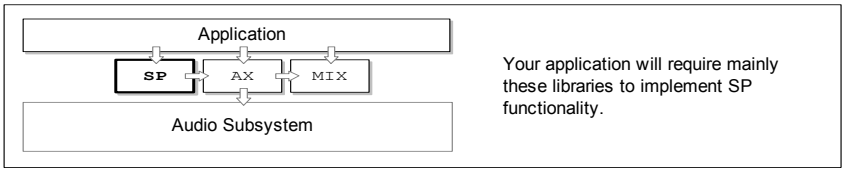
The final piece in the AX Sound Pipeline is the SP runtime library. This library works in conjunction with AX to provide the following:

- An abstraction layer for convenient access to sound effect parameters
- Automatic address resolution for samples in main memory
- Automatic initialization of AX voices to simplify playback of sound effects

The SP runtime library assumes the following:

- The application loads Sound Table data into main memory from an SPT file
- The application loads the associated Sound Data into main memory from an SPD file
- The application has been compiled with the associated C header file generated by `sndconv`

Figure 4–1 Relationship between Various Audio Libraries



4.2 Data Abstraction

The SP runtime library defines the `SPSoundTable` data structure to describe each sound effect:

Code 4–1 SPSoundEntry Data Structure

```
typedef struct
{
    u32             type;

#define SP_TYPE_ADPCM_ONESHOT   0
#define SP_TYPE_ADPCM_LOOPED    1
#define SP_TYPE_PCM16_ONESHOT   2
#define SP_TYPE_PCM16_LOOPED    3
#define SP_TYPE_PCM8_ONESHOT    4
#define SP_TYPE_PCM8_LOOPED     5

    u32             sampleRate;
    u32             loopAddr;
    u32             loopEndAddr;
    u32             endAddr;
    u32             currentAddr;
    SPAdpcmEntry    *adpcm;

} SPSoundEntry;
```

Each entry describes a sound effect in the associated SPD construct.

The SPInitSoundTable() function does the following:

- Resolves the offset values in loopAddr, loopEndAddr, endAddr, and currentAddr against the base address of the SPD construct in main memory
- Initializes the SPAdpcmEntry *adpcm pointer with the appropriate ADPCM descriptor entry at the end of the SPT aggregate

See “SP API” on page 39 for more details on the SPInitSoundTable() function.

Code 4–2 SpAdpcmEntry Data Structure

```
typedef struct
{
    AXPBADPCM       adpcm;
    AXPBADPCMLOOP   adpcmloop;

} SPAdpcmEntry;
```

Each ADPCM-encoded sound effect has a corresponding entry of type SPAdpcmEntry. The constituent data structures, AXPBADPCM and AXPBADPCMLOOP are defined by the AX library, and are given in Code 4–3 for completeness:

Code 4–3 AX ADPCM Data Structures

```
typedef struct _AXPBADPCM
{

    u16     a[8][2];            //  coef table a1[0],a2[0],a1[1],a2[1]....

    u16     gain;               //  gain to be applied (0 for ADPCM, 0x0800 for PCM8/16)

    u16     pred_scale;         //  predictor / scale combination (nibbles, as in hardware)
    u16     yn1;                //  y[n - 1]
    u16     yn2;                //  y[n - 2]

} AXPBADPCM;

typedef struct _AXPBADPCMLOOP
{
    u16     loop_pred_scale;    //  predictor / scale combination (nibbles, as in hardware)
    u16     loop_yn1;           //  y[n - 1]
    u16     loop_yn2;           //  y[n - 2]

} AXPBADPCMLOOP;
```

The application must load the SPT file into memory verbatim. A pointer of type `SPSoundTable` must be assigned to the beginning of this data.

Code 4–4 SPSoundTable Data Structure

```
typedef struct
{

    u32             entries;
    SPSoundEntry    sound[];

} SPSoundTable;
```

Note: The first member is an integer specifying the total number of sound effects recorded in the SPT table. In addition, the start of the `SPAdpcmEntry` data is implicit: the `SPInitSoundTable()` function simply assigns the start pointer to the end of the `SPSoundEntry` structures.

The `SPInitSoundTable()` function initializes each `SPSoundEntry` structure sequentially. As it encounters `ADPCM`-encoded sound effects, it assigns the `SPAdpcmEntry *adpcm` pointer to subsequent `SPAdpcmEntry` entries.

4.3 SP API

This section describes the AX Sound Pipeline runtime library functions.

4.3.1 The SPInitSoundTable Function

Code 4–5 SPInitSoundTable()

```
#include <revolution/sp.h>

void SPInitSoundTable(SPSoundTable *table, u8 *samples, u8 zerobuffer);
```

The `SPInitSoundTable` function does the following:

- Resolves the main memory addresses for each entry in the table against `*samples`
- Initializes `ADPCM` information pointers for `ADPCM`-encoded sound effects

Prior to calling this function, the application must allocate memory for the sound table and load the data from an SPT file.

Arguments:

`SPSoundTable *table`

Pointer to an instance of the `SPSoundTable` structure. Each entry in `SPSoundTable` corresponds to a sound effect in an SPD file.

`u8 *samples`

Pointer to a main memory address (either `MEM1` or `MEM2`) where the corresponding SPD data has been stored. The SPD data must be 32-byte aligned.

`u8 *zerobuffer`

Not used. Be sure to specify NULL.

Return Values:

None.

4.3.2 The SPGetSoundEntry Function

Code 4–6 SPGetSoundEntry()

```
#include <revolution/sp.h>

SPSoundEntry * SPGetSoundEntry(SPSoundTable *table, u32 index);
```

For the given sound table instance and sound effect index, the `SPGetSoundEntry` function returns a pointer to the corresponding SP sound table entry.

Arguments:

`SPSoundTable *table`

Pointer to an array of `SPSoundTable` structures. Each `SPSoundTable` entry corresponds to a sound effect in an SPD file.

`u32 index`

Sound effect ID enumerated in a Sound Pipeline header file generated by `sndconv`.

Return Values:

Pointer to an SP sound table entry. If `index` is not within the valid range of entries, this function returns NULL.

4.3.3 The SPPrepareSound Function

Code 4–7 SPPrepareSound()

```
#include <dolphin/sp.h>

void SPPrepareSound(SPSoundEntry *sound, AXVPB *axvpb, u32 sampleRate);
```

The `SPPrepareSound` function prepares a voice for playback of the specified sound effect.

Note: This function does not start the playback. The application must explicitly set the voice state to `AX_PB_STATE_RUN` by using `AXSetVoiceState()`.

Arguments:

`SPSoundEntry *sound`

Pointer to an entry in an SP sound table. This function will `ASSERT` a failure if `sound` is NULL.

`AXVPB *axvpb`

Pointer to an AX voice parameter block.

Note: The application must acquire a voice from AX before calling this function. This function will `ASSERT` a failure if `sound` is NULL.

`u32 sampleRate`

Integer specifying the frequency, in Hz, at which the sound must be played. To play the sound with the default frequency, use `sound->sampleRate` for this argument.

Return Values:

None.

This function copies the parameters of the sound from the SPSoundEntry structure into the appropriate locations in the AX voice parameter block.

4.3.4 The SPPrepareEnd Function

Code 4–8 SPPrepareEnd()

```
void SPPrepareEnd(SPSoundEntry *sound, AXVPB *axvpb);
```

The SPPrepareEnd function marks a looped voice for termination.

The voice is specified by:

- A sound effect entry in an SP sound table
- A voice acquired from AX, currently playing this sound effect

This function is useful for terminating a looped sound effect that is already playing. It simply clears the loop flag of the voice and resets the end address of the sound effect (if necessary). Thus, the sound effect plays to the end, and then stops.

This function has no effect on non-looped sound effects.

The function must be called only after a looped sound effect has started.

In other words, calling this function immediately after SPPrepareSound() (within the same audio frame) does not work because the synchronization bits used for SPPrepareEnd() have precedence over those asserted by SPPrepareSound(). Therefore, the initialization performed by SPPrepareSound() is lost, and the sound plays incorrectly.

If you want to play a looped sound one time, clear the *loop* flag of the voice, and reset its end address manually after calling SPPrepareSound(). Code 4–9 is an example.

Code 4–9 Clearing a Voice's Loop Flag and Resetting its End Address Manually

```
static void foo(SPSoundEntry *sound, AXVPB *axvpb, BOOL one_shot);
{
        .
        .
        .
        SPPrepareSound(sound, axvpb, sound->sampleRate);
        if (TRUE == one_shot)
        {
                // This is a looped sound, but we want to play it as a one-shot.
                // So, we must revise the loop flag and end address.
                axvpb->pb.addr.loopFlag = AXPBADDR_LOOP_OFF;
                axvpb->pb.addr.endAddressHi = (u16)(sound->endAddr >> 16);
                axvpb->pb.addr.endAddressLo= (u16)(sound->endAddr & 0xFFFF);
        }
        .
        .
        .

} // end foo()
```

Note: You do not have to assert synchronization bits for the revised loopFlag and endAddressHi/Lo parameters, because the preparatory SPPrepareSound() function already does so.

Arguments:

`SPSoundEntry *sound`

Pointer to an entry in an SP sound table.

`AXVPB *axvpb`

Pointer to an AX voice parameter block.

Return Values:

None.

4.4 Using SP

4.4.1 Source Code

The source code for the SP library can be found in the Nintendo Revolution SDK installation directory at the following location:

`/build/libraries/sp/`

The include file can be found at the following location:

`/include/revolution/sp.h`

Note: The SP runtime library is dependent on the AX header file (`ax.h`), as well.

4.4.2 Loading the SP Sound Table

The SP sound table must reside in main memory. Loading the file from disc is straightforward.

Code 4–10 Loading the SPT File into Main Memory

```
#include <revolution.h>
#include <revolution/sp.h>
#include <revolution/mem.h>

#define mROUNDUP32(x)   (((u32)(x) + 32 - 1) & ~(32 - 1))
MEMHeapHandle hExpHeap;
void           *arenaMem2Lo;
void           *arenaMem2Hi;
static SPSoundTable *sp_table;

static void *load_file(char *path, u32 *length)
{
    DVDFileInfo dvdFileInfo;
    u32         roundLength;
    void        *buffer;

        // open file
        DVDOpen(path, &dvdFileInfo));

        // get length, round up to next 32 Bytes
        roundLength= mROUNDUP32(DVDGetLength(&dvdFileInfo));

        // allocate memory

        buffer = MEMAllocFromExpHeapEx(hExpHeap, roundlength, 32);

        // read data; assume DVD auto-invalidate is ON!
        DVDRead(&dvdFileInfo, buffer, (s32)(roundLength), 0);

        *length = roundLength;

        return(buffer);

} // end load_file()

.
.
.
{
  u32 length;

    //initialize Exp Heap on MEM2
    arenaMem2Lo = OSGetMEM2ArenaLo();
    arenaMem2Hi = OSGetMEM2ArenaHi();
    hExpHeap    = MEMCreateExpHeap(arenaMem2Lo, (u32)arenaMem2Hi - (u32)arenaMem2Lo);

    sp_table = (SPSoundTable *)load_file("test.spt",  &length);

}
```

Note: This example allocates memory for the SPT data at runtime using the MEM library. If you want to allocate memory statically, be sure that the buffer is 32-byte aligned.

Code 4–11 Statically Allocating Memory

```
static u8 *buffer[SPT_SIZE] ATTRIBUTE_ALIGN(32);
```

Also be sure that your buffer is rounded up to the next multiple of 32 bytes.

4.4.3 Loading the SPD file into Main Memory

In the following example, the `load_file` function (described previously) loads an SPD file from disc into memory (`spd_buffer`) allocated using the MEM library.

Code 4–12 Loading SPD Files into Main Memory

```
#include <revolution.h>
#include <revolution/sp.h>
#include <revolution/mem.h>

MEMHeapHandle  hExpHeap;
void          *arenaMem2Lo;
void          *arenaMem2Hi;
void          *spd_buffer;
  .
  .
  .
{
  u32   length;
    .
    .
    .
    // initialize Exp Heap on MEM2
    arenaMem2Lo = OSGetMEM2ArenaLo();
    arenaMem2Hi = OSGetMEM2ArenaHi();
    hExpHeap    = MEMCreateExpHeap(arenaMem2Lo, (u32)arenaMem2Hi - (u32)arenaMem2Lo);

    // use load_file() function from previous example
    spd_buffer = load_file("test_sfx.spd", &length);

}
```

4.4.4 Initializing the SP Sound Table

When the sound table and data have been loaded, you can initialize the SP library, as shown in Code 4–13.

Code 4–13 SP Sound Table Initialization

```
#include <revolution.h>
#include <revolution/sp.h>
#include <revolution/mem.h>

static SPSoundTable *sp_table;
static u8           *sp_data;.
.
{
  .
  .
  .
  // load SPT data (see examples above)
  .
  .
  .
  // load SPD data (see examples above)
  .
  .
  .

  // Here we go!
  SPInitSoundTable(sp_table, sp_data, NULL);

  // that's it!

}
```

4.4.5 Preparing a Sound Effect for Playback

Code 4–14 Playing a Sound Effect

```
#include <revolution.h>
#include <revolution/sp.h>

#include "test_sfx.h"



static AXVPB *voice;

static SPSoundTable *sp_table;
static SPSoundEntry *sp_entry;

.
.
.
{
    // Initialize AX and Mixer (see AX and MIX documentation)

    // Load SPT file (see previous examples)

    // Load SPD data (see previous examples)

    // Initialize SPT table
```

```
    // Get sound table entry for given sound effect index
    // NOTE: The index is defined in the test_sfx.h header file
    sp_entry = SPGetSoundEntry(sp_table, SFX_BLAMMO);

    // Now acquire a voice!
    voice = AXAcquireVoice(15, NULL, 0);

    if (voice)
    {
        // play at default sampling frquency
        SPPrepareSound(sp_entry, voice, sp_entry->sampleRate);

        // setup a mixer channel for this voice
        MIXInitChannel(voice, 0, 0, -960, -960, 64, 127, 0);

        // Set voice state to run!
        AXSetVoiceState(voice, AX_PB_STATE_RUN);

     }

}
```

Note: The sound effect enumeration, `SFX_BLAMMO`, is defined in the header file associated with the SPT and SPD data (`test_sfx.h`).

A voice must be acquired before a sound effect is prepared for playback. Playback must be also started manually with a call to `AXSetVoiceState()`.

4.4.6 Preparing a Looped Effect for Termination

Code 4–15 Stopping a Looped Sound Effect

```
#include <revolution.h>
#include <revolution/sp.h>

#include "test_sfx.h"


static AXVPB *voice;

static SPSoundTable *sp_table;
static SPSoundEntry *sp_entry;

.
.
.
{
    // Initialize AX and Mixer (see AX and MIX documentation)

    // Load SPT file (see previous examples)

    // Load SPD data (see previous examples)

    // Initialize SPT table

    // Start playing a looped sound effect (see previous examples)


    .
    .
    .

    if (TRUE == please_stop_this_sound)
    {
        // Refresh voice parameter data from SP sound entry
        SPPrepareEnd(sp_entry, voice);
    }

}
```

Once a looped sound effect is playing, you can easily terminate the sound, using the `SPPrepareEnd()` function. This function re-initializes the loop state information of the voice with values from the table entry of the sound effect, thus causing the voice to stop. The AX user application can then free the voice.

Note: We recommend applying a volume envelope or a fade-out before stopping a voice, to prevent discontinuity.

TM and ® are trademarks of Nintendo.
Dolby, Pro Logic and the Double-D symbol are trademarks of Dolby Laboratories.
IBM is a trademark of International Business Machines Corporation.
Roland GS Sound Set is a trademark of Roland Corporation U.S.
All other trademarks and copyrights are property of their respective owners.

© 2006-2008 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.