Online Technical Article, January 2014
Edinburgh Napier University - Playstation 3 Development

# Lesson 9
# Low-level Sound Playback

## Playstation 3 Development

Sam Serrels and Benjamin Kenwright[1]

**Abstract**
This tutorial will cover foundational audio code to output raw sound samples from the Playstation 3, with explanations of the low-level Sony sound libraries. Higher-level libraries will be introduced and concepts such as audio mixing and routing will be touched upon within this tutorial. Additionally, this tutorial will cover the parsing of Wav audio files and the code to play them back will be created, which builds upon the basic code covered in the first section.

**Keywords**
Sony, PS3, Development, Tutorial, PlayStation, PPU, Programming, Sound, Music, Mixer

[1] *Edinburgh Napier University, School of Computer Science, United Kingdom*: b.kenwright@napier.ac.uk

## Contents



## Preface

**About the Edinburgh Napier University Game Technology Playstation 3 Development Lessons** Edinburgh Napier University Game Technology Lab is one of the leading game teaching and research groups in the UK - offering students cutting edge facilities that include Sony's commercial development kits. Furthermore, within the Edinburgh Napier Game Technology group are experienced developers to assist those students aspiring to releasing their own games for PlayStation. Students have constant access to he Sony DevKits and encourage enthusiastic students to design and build their own games and applications during their spare time [1].

**Previous Tutorials** This tutorial assumes you have read the previous tutorials on compiling and deploying applications to the PS3.

## 1. Introduction

**Sound** Audio is a quintessential part of the experience of video games, any game is never complete without good audio to immerse the players. Short Sound effects can be used to promote interactivity and can make the game more intuitive, whereas catchy and memorable background music can remain in your players head for almost forever.
The audio capabilities of the Playstation 3 are broad, as to be expected it starts with the lowest of levels, with near hardware access to the audio decoders. Fortunately, there are other higher level solutions available to use, but with abstraction comes limitations.

## 2. Playstation 3 Sound

### 2.1 Playstation 3 Sound Libraries

**The Sound Libraries of The PS3** There a few different libraries that deal with Sound on the Playstation 3, they are:

**LibAudio**
The lowest level of audio, handles the audio hardware. Everything else is built up from this.

**LibMixer**
Library for managing audio channels. Tasks such as per channel volume and routing are handled. This combined with Libaudio is enough to get a very rudimentary audio system working. e.g playing Sine Waves

**Simple Sound Player**
Part of LibMixer, allows playback of WAV files. Has functions for pitch, reverse, and looping playback.

**libsnd3**
: MIDI playback

**libsynth2**
: Software synthesizer, mainly used for Playstation 2 audio backwards compatibility

**Multistream**
: A high level Audio system.

**MSMP3**
: MultiStream compatible MP3 decoding

**SCREAM**
: **SCR**iptable **E**ngine for **A**udio **M**anipulation (SCREAM) A Sound Effect Bank player, imagine playSound(*"bullet"*); or playSound(*"jumpSFX"*);

**LibAudio and LibMixer** In this Tutorial we will be dealing with the lowest level audio libraries, to implement a super simple "noise generator" class. After this is established, the code to load, parse and play uncompressed WAV files will be created. In future tutorials playback of other sound formats will be dealt with, as it requires a substantial amount of additional code.

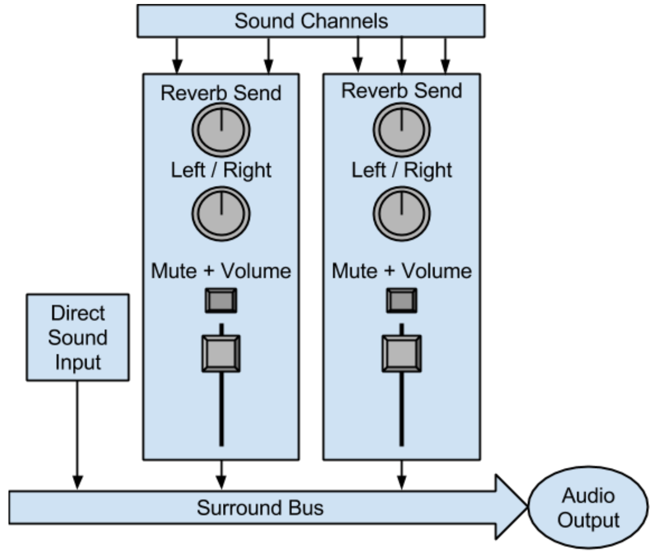


**Figure 1. Audio Mixer with channel strips** - Some Parameters omitted from channels

## 2.2 Playstation 3 Audio Features

**Channel Strips** Like a real-world mixer, audio streams can be processed as individual channels with individual effects placed upon them, then they are combined into one output stream. These effects channels are called "Channel Strips". Unlike a real-world mixer, any number of inputs can go into one channel strip, and the effects are applied to them all. You could have one strip that takes in 8 channels of audio. Furthermore an (almost) unlimited amount of channel strips can be generated. See Fig:1

**Direct Sound Input** Audio samples can be directly inserted into the Audio bus, bypassing the channel strips. This is what we will be doing in this Tutorial. This takes the form of sending raw audio samples to a desired output channel. If you need to mix any channels or apply any effects, you must do this in your own code, or use the Mixer.

**Audio output formats** The PS3 audio outputs include: Optical Digital Audio, Digital Audo over HDMI, and analogue audio through the AV breakaway cable. Surround sound of up to 8 channels (7.1) is supported over the digital outputs, in a variety of encoding (DTS,AC-3,PCM). The AV out cable only supports 2 channel analogue stereo.

**Devkit Audio output formats** The devkit comes equipped with 4 analogue audio output jacks, which allow debugging of the surround sound channels without having to have a separate digital audio decoder. The configuration of these jacks is undocumented, but it is probably: (Front, Center, Back, Sub). This may be used to debug the multiheaphone mode.

**Selecting an output format** The output format is a combination of two factors: the amount of channels and the encoding codec. We don't actually have the ability to *set* the output format, we must enquire to see which formats are available and then pick from them. Availability depends on which cable is plugged in, system audio detection and user settings. Choosing a format and configuration that isn't available will result in a crash.
The best practise is to enquire about each decoder type, PCM, DTS and AC3. The system will report back the amount of available channels for each, if one is unavailable the system will report back 0 channels for it. It is assumed that a minimum of two PCM channels are always available. Once you have the acquired the capabilities, you must decide which to use. The choice of which encoder to use is something that you will have to research to find out which is best for your requirements. For the amount of channels, if your game supports a surround sound setup, but one isn't available, you can down mix two a lower number of channels.

# 3. Foundation Sound Code

This code selects the apporopriate output format, and then plays back a sequence of tones through the various output channels (I.e 5.1), which will be down-mixed to 2 channels in most cases.

**Listing 1.** Sound playback File

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/timer.h>
#include <cell/audio.h>
#include <cell/mixer.h>
#include <cell/sysmodule.h>
#include <sysutil/sysutil_sysparam.h>

#define DBG_HALT { exit (1); }
```

```
// Calls suplied function on assert fail, then call DBG_HALT
#define DBG_ASSERT_FUNC(e,f){if(!(e)){f;DBG_HALT;}}

int SoundCallback(void *arg0,uint32_t index,uint32_t samples);
int SystemAudioUtilitySet6ch(void);

static const CellSurMixerConfig g_SurMixerConfigS0 = {
 400,      /* thread priority */
 0,        /* the number of 1ch channel strip */
 0,        /* the number of 2ch channel strip */
 0,        /* the number of 6ch channel strip */
 0         /* the number of 8ch channel strip */
};

// Test tone generator data ---------------
// The base pitch to work from
static float g_Pitch0 = 220.0;
// The actual Saw Wave data to output
static float g_Saw[8] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
// Acending Pitch data to apply to the output data.
static float g_SawAdd[8] =  {g_Pitch0 * 2.0 / 48000.0,
 g_Pitch0 * pow(2.0, 200.0 / 1200.0) * 2.0 / 48000.0,
 g_Pitch0 * pow(2.0, 400.0 / 1200.0) * 2.0 / 48000.0,
 g_Pitch0 * pow(2.0, 500.0 / 1200.0) * 2.0 / 48000.0,
 g_Pitch0 * pow(2.0, 700.0 / 1200.0) * 2.0 / 48000.0,
 g_Pitch0 * pow(2.0, 900.0 / 1200.0) * 2.0 / 48000.0,
 g_Pitch0 * pow(2.0, 1100.0 / 1200.0) * 2.0 / 48000.0,
 g_Pitch0 * pow(2.0, -2400.0 / 1200.0) * 2.0 / 48000.0};

// Channel Data -----------------
static unsigned int channelSequence[8] = {0, 1, 5, 4, 2, 7, 6, 3};
static const char *channelName[8] = {
 "Left",
 "Right",
 "SurRight",
 "SurLeft",
 "Center",
 "ExtRight",
 "ExtLeft",
 "LFE"
};

static unsigned int activeChannel = 0;
unsigned int channelSequenceIndex = 0;

int SystemAudioUtilitySet6ch(void) {
 int ret, num, ch_pcm, ch_ac3,ch_dts ;
 CellAudioOutConfiguration conf;
 CellAudioOutState a_state;

 // Find out how many audio channels are available
 // with which audio technology.

 // Linear PCM, is always available with at least 2 channels.
 ch_pcm = cellAudioOutGetSoundAvailability(
   CELL_AUDIO_OUT_PRIMARY,
   CELL_AUDIO_OUT_CODING_TYPE_LPCM,
   CELL_AUDIO_OUT_FS_48KHZ, 0);

 // How many Dolby AC-3 channels available?
 // If this encoder can not be used, 0 is returned.
 ch_ac3 = cellAudioOutGetSoundAvailability(
   CELL_AUDIO_OUT_PRIMARY,
   CELL_AUDIO_OUT_CODING_TYPE_AC3,
   CELL_AUDIO_OUT_FS_48KHZ, 0);

 // How many Dolby DTS channels available?
 // If this encoder can not be used, 0 is returned.
 ch_dts = cellAudioOutGetSoundAvailability(
   CELL_AUDIO_OUT_PRIMARY,
   CELL_AUDIO_OUT_CODING_TYPE_DTS,
   CELL_AUDIO_OUT_FS_48KHZ, 0);

 // If audio is being outputted via SPDIF, The PS3 can not
 // obtain the capabilities of the output automatically.
 // This must be set by the user in the system sound settings.

 // Which encoder to use depends on the desired use.
 // If zero-lag audio is needed, PCM is the best encoder
 // This application uses a 5.1ch layout.
 // Libaduio supports 8ch, however this uses only 6ch.

 // Chose an output mode ---------

 // The target structure must be zero cleared.
 memset(&conf, 0, sizeof(CellAudioOutConfiguration));

 if( ch_pcm >= 6 ) {
  // If 6ch of linear PCM is available, we use this.
  conf.channel = 6;
  conf.encoder = CELL_AUDIO_OUT_CODING_TYPE_LPCM;
  printf("Audio Encoder: PCM, Channels: 6\n");
 }else if( ch_ac3 >= 6 ) {
  // else if AC-3 is available, we use this.
  conf.channel = 6;
  conf.encoder = CELL_AUDIO_OUT_CODING_TYPE_AC3;
  printf("Audio Encoder: AC-3, Channels: 6\n");
 }else if( ch_dts >= 6 ) {
  // else if DTS is available, we use this.
  conf.channel = 6;
  conf.encoder = CELL_AUDIO_OUT_CODING_TYPE_DTS;
  printf("Audio Encoder: DTS, Channels: 6\n");
 }else {
  // else if neither is available, use 2ch of linear PCM.
  conf.channel = 2;
  conf.encoder = CELL_AUDIO_OUT_CODING_TYPE_LPCM;
  printf("Encoder: PCM, 2 channels, Downmixed from 6\n");
  // We must then downmix the multiple channels to stereo.
  // Luckily there is a built in downmixer
  conf.downMixer =
   CELL_AUDIO_OUT_DOWNMIXER_TYPE_A;
 }

 // Apply the selected audio settings ------

 // This may take some time if the audio system is busy
 // Keep looping and trying to apply until it works
 for(num=0; num< 400; num++ ) {
  ret = cellAudioOutConfigure(
     CELL_AUDIO_OUT_PRIMARY, &conf, NULL, 0);
  if( ret == CELL_OK ) {
   break; //Successs!
  }
  else if(ret! = CELL_AUDIO_IN_ERROR_CONDITION_BUSY){
   DBG_HALT;
   break; //An error!
  }
  sys_timer_usleep(5000);
 }
 DBG_ASSERT_FUNC(( ret == CELL_OK ),
  printf("#ERR cellAudioOutConfigure() failed (0x%x).\n", ret));

 // Check to see if the audio system is now ready. Loop like before.
 for(num=0; num< 400; num++ ) {
  ret = cellAudioOutGetState(
     CELL_AUDIO_OUT_PRIMARY, 0, &a_state);
  if( ret == CELL_OK ) {
   if(a_state.state ==
     CELL_AUDIO_OUT_OUTPUT_STATE_ENABLED){
    break; // Audio is enabled and ready
   }
  }
  else if(ret!=CELL_AUDIO_OUT_ERROR_CONDITION_BUSY)
  {
   DBG_HALT;
   break; //An error!
  }
  sys_timer_usleep(5000);
 }
 return ret;
}

int main(void) {
 int err;
```

```

    //Load the Audio system module
    err = cellSysmoduleLoadModule(CELL_SYSMODULE_AUDIO);
    DBG_ASSERT_FUNC( (err == CELL_OK),
      printf("cellSysmoduleLoadModule() : %x\n", err));

    // Configure the audio output mode
    err = SystemAudioUtilitySet6ch();
    DBG_ASSERT_FUNC((err >= 0),
      printf("SystemAudioUtilitySet6ch() error : %x\n", err));

    // Initialize audio system
    err = cellAudioInit();
    DBG_ASSERT_FUNC((err >= 0),
      printf("cellAudioInit() : %x\n", err));

    // Create a mixer object
    err = cellSurMixerCreate(&g_SurMixerConfigS0);
    DBG_ASSERT_FUNC((err >= 0),
      printf("cellSurMixerCreate() error : %x\n", err));

    //Link our mixer callback function to the mixer.
    err = cellSurMixerSetNotifyCallback(SoundCallback, NULL);
    DBG_ASSERT_FUNC((err >= 0),
      printf("cellSurMixerSetNotifyCallback() error : %x\n", err));

    activeChannel = channelSequence[channelSequenceIndex];

    //Starts The mixer
    err = cellSurMixerStart();
    DBG_ASSERT_FUNC((err >= 0),
      printf("cellSurMixerStart() error : %x\n", err));

    //Continuously play sounds
    while (1) {
      activeChannel = channelSequence[channelSequenceIndex];
      printf("Channel[%i]: %s \n",
        activeChannel, channelName[channelSequenceIndex]);
      sys_timer_usleep(800 * 1000);
      channelSequenceIndex = (channelSequenceIndex + 1) & 7;
    }

    printf("Exiting\n");

    return 0;
}

// Function used as a callback from the mixer
// __attribute__((unused)) is a compiler macro, it informs the
// compiler that you expect a variable to be unused
int SoundCallback( void *arg0 __attribute__((unused)),
      uint32_t index __attribute__((unused)), uint32_t samples)
{
    int err;
    float buff[1024];

    for (unsigned int i = 0; i < samples; i++)
    {
      g_Saw[channelSequenceIndex] +=
          g_SawAdd[channelSequenceIndex];
      if (g_Saw[channelSequenceIndex ] > 1.0){
        g_Saw[channelSequenceIndex ] -= 2.0;
      }
      buff[i] = 0.2 * g_Saw[channelSequenceIndex ];
    }
    err =cellSurMixerSurBusAddData(activeChannel,0,buff,samples);
    DBG_ASSERT_FUNC((err >= 0),
        printf("cellSurMixerSurBusAddData() error : %x\n", err));
    return 0 ;
}
```

### 3.1 The purpose of this Code

So we have super simple sounds being played, is this useful for games an applications? Not really, at least not in this state. You could have an array of distinct sound waves that you could playback as sound effects, if you want to go for that *super* retro feel.

Realistically you will want something more for your games, playback of either proper audio or more interesting polyphonic sound effects. This can be achieved with multiple methods, you could find/write your own audio decoder, use the Sony Libraries or use a 3rd party audio solution. The point is that any of these routes will all use the same code written in this tutorial as the foundation for all audio playback.

With this code you have the ability to send out raw audio samples, now all you have to do it find a source for those samples.

## 4. Wav Playback

**Simple Sound Player** The Simple Sound player is not a library in it's own right, it is just a couple of functions and structures that live within libMixer that deal with parsing the pcm audio data of Wav files. We still have to load a Wav into memory and do some parsing of the file structure ourselves, but we don't have to manually do any sort of loop where we send wav samples to the mixer. In theory we could do this ourselves, and it wouldn't be too difficult, but as SSP exists we may as well use it.

### 4.1 The Wav File

A Wav(Waveform Audio) file is the most basic audio container there is, however it still contains a small amount of Metadata. The container format used by Wavs is RIFF (Resource Interchange File Format), while the audio data is just pcm with no compression, i.e no compression codec.

**Resource Interchange File Format** RIFF files are like a simple byte level version of XML. It is sectioned into chunks, the first four bytes of a chunk contain the chunk name, the next 4 bytes contain the size of the data section, and the remaining bytes are the data section itself. There are a few other data bits around the file for padding and the Endianness(Big or Little) or the data can vary. The file can contain multiple chunks, and chunks can be nested in other chunks. The beauty of RIFF is that new chunks can be added without breaking compatibility. If an application doesn't understand how to deal with a certain chunk, it ignores it.

**Wav Chunk structure** The structure of a Wav can vary, but the standard minimum is this:

**RIFF** - *Container chunk*

  **WAVE** - *Wave data chunk*

    **FMT** - *Sound Format**

    **DATA** - *Actual sound data*

  **/WAVE**

**/RIFF**

*Sample rate, Bitrate, Number of channels...

# 5. Wave Playback Code

## 5.1 wavfile.h

The Wave file class is generic code not specific to the PS3.

**Listing 2.** Wav playback, wavfile.h

```
#pragma once
#include <stdio.h>

class Wave
{
public:
  //File on disk
  static FILE* file;
  // File data in memeory
  static char* buffer;
  // reported FileSize from filesystem
  static long filebytesize;
  //Index of the wave data chunk in the file
  static short* wavetop;
  //reported size of the data chunk
  static uint32_t waveByteSize;

  //Load in a Wave File, saves data in to the statics above.
  static int readWavfile(const char * filename);

  // Swap Endinan-ess of an uint32_t
  static uint32_t rev32(uint32_t in);

  // returns the Index of the specified target chunk in a Wav file
  static int seekWavChunk(
     char *top,       // The Wav file
     unsigned int bytesize,  // the size of the Wav File
     char *target     // Which Section of the Wav fiel to load
   );

  //Reads the reported chunk size from the specified wav chunk
  static uint32_t getWavChunkSize(
     char *top,       //The Wave File
     unsigned int offset   //Index of chunk to get filesize for
   );
};
```

## 5.2 wavfile.cpp

**Listing 3.** Wav playback, wavfile.cpp

```
#include "wavfile.h"
#include <cell/cell_fs.h>
#include <stdlib.h>

FILE* Wave::file = NULL;
char* Wave::buffer = NULL;
long Wave::filebytesize = 0;
short* Wave::wavetop = NULL;
uint32_t Wave::waveByteSize = 0;

//Load in a Wave File, saves data in to the statics above.
int Wave::readWavfile(const char * filename)
{
   printf(" loading %s\n", filename);

   // Look for file
   file = fopen(filename, "rb");
   if  (file != NULL)
   {
      // Load file attributes
      fseek(file, 0, SEEK_END);
      filebytesize = ftell(file);
      if (filebytesize == 0){
         printf("seek error.\n");
         return (-1);
      }
      printf("Filebytesize is %ld\n", filebytesize);
      fseek(file, 0, SEEK_SET);

      // Reserve memory
      buffer = (char *)malloc(filebytesize);
      if (buffer == NULL){
         printf("malloc error.\n");
         return (-1);
      }

      // Copy file into memory
      if (fread(buffer, filebytesize, 1, file) != 1){
         printf(" ## ERR fread fp1 \n");
         fclose(file);
         return (-1);
      }
      printf(" %s  %ld bytes read \n", filename, filebytesize);

      // Done with the data , close the file.
      fclose(file);

      //Get the Wav attributes of the loaded file
      const char strData[] = {'d', 'a', 't', 'a'};
      int i = seekWavChunk(buffer, filebytesize, (char *)strData);
      if (i > 0){
         wavetop = (short *)(buffer+i+4);
         waveByteSize = getWavChunkSize(buffer, (unsigned int)i);
         printf("  waveByteSize = %d\n", waveByteSize);
      } else {
         return (-1);
      }
   }else{
      printf(" couldn't find %s\n", filename);
   }

   return 0;
}

// Swap Endinan-ess of an uint32_t
uint32_t Wave::rev32(uint32_t in)
{
   unsigned char *s1, *d1;
   uint32_t out = 0;

   d1 = (unsigned char *)&out;
   s1 = (unsigned char *)&in;

   *d1++ = *(s1 + 3);
   *d1++ = *(s1 + 2);
   *d1++ = *(s1 + 1);
   *d1++ = *s1;

   return (out);
}

//returns the Index of the specified target chunk in a Wav file
int Wave::seekWavChunk
(char *top, unsigned int bytesize, char *target)
{
   int            i;
   char           chunk[4];
   uint32_t       u32;
   unsigned char  *up;
   unsigned int   index = 0;
   unsigned int   skip = 0;
   char           *cp;
   const char strRIFF[] = {'R','I','F','F'};
   const char strWAVE[] = {'W','A','V','E'};

   //Check to see if file begins with 'RIFF'
   for (i = 0; i < 4; i++){
      if (top[index++] != strRIFF[i]){
         return (-1);
      }
   }

   //The next 4 bytes contatin the file size,
```

```
    //Disgard this as we calculate this ourselves
    up = (unsigned char *)&u32;
    for (i = 0; i < 4; i++){
        *up++ = top[index++];
    }

    // The next 4 bytes should be 'WAVE'
    for (i = 0; i < 4; i++){
        if (top[index++] != strWAVE[i]){
            return (-1);
        }
    }

    while (1){
        //Fill up the 4 byte chunk buffer
        for (i = 0; i < 4; i++){
            chunk[i] = top[index++];
        }

        //The 1st 4 bytes of a chunk is its ID, does it match the target?
        for (i = 0; i < 4; i++){
            if(chunk[i] != target[i]){
                break; //didn't match, break
            }
        }

        if (i == 4){
            return index; // We found our chunk, return the Index
        }

        if (index >= bytesize){
            return (-1); //reached the end of the file, return error
        }

        cp = (char*)&u32;
        for (i = 0; i < 4; i++){
            *cp++ = top[index++];
        }

        if (index >= bytesize){
            return (-1); //reached the end of the file, return error
        }
#if _ENDIAN==BIG_ENDIAN
        u32 = rev32(u32);
#endif
        index += u32;

        if (index >= bytesize){
            return (-1); //reached the end of the file, return error
        }

        skip++; //increment our skip counter
        if (skip > 16){
            break; //We have skipped too many times, return error
        }
        //Loop round again
    }
    return (-1);
}

    //Reads the reported chunk size from the specified wav chunk
uint32_t Wave::getWavChunkSize(char *top, unsigned int offset)
{
    uint32_t      u32;
    unsigned char  *up = (unsigned char *)&u32;
    int           i;

    for (i = 0; i < 4; i++){
        *up++ = top[offset++];
    }
#if _ENDIAN==BIG_ENDIAN
        u32 = rev32(u32);
#endif
    return u32;
}
```

## 5.3 main.cpp

The new Main code is very silimar to the original code, the set-up function is unchanged. The difference is that we don't have to send any data to the mixer, as SSP does that. We also need to set-up a channel strip this time around, as SSP can only connect to the mixer through them. Finally we load a Wav through our new Wavfile class, parse the attributes, send them to SSP, and tell it to play.

The Asserts have been removed from this listing for clarity. Wherever there is a "err=" there should be this following it:

```
DBG_ASSERT_FUNC( (err >= 0) , printf("Error : %x\n", err) );
```

**Listing 4.** Wav playback, main.cpp

```
#include "wavfile.h"
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/paths.h>
#include <sys/timer.h>

#include <cell/audio.h>
#include <cell/mixer.h>

#include <cell/sysmodule.h>
#include <sysutil/sysutil_sysparam.h>

//#define DBG_HALT { __asm__ volatile( "trap" ); }
#define DBG_HALT { exit (1); }
// Calls the suplied function on assert fail, then call DBG_HALT
#define DBG_ASSERT_FUNC( exp, func) { if ( !(exp) ) {func; DBG_HALT;} }

int SystemAudioUtilitySet6ch(void);

static const CellSurMixerConfig g_SurMixerConfigS0 = {
  400,      /* thread priority */
  0,        /* the number of 1ch channel strip */
  1,        /* the number of 2ch channel strip */
  0,        /* the number of 6ch channel strip */
  0         /* the number of 8ch channel strip */
};

//  Handle of Surround Mixer to which SSP will be connected
static CellAANHandle mixerHandle;

//  Port number of Channel Strip to which SSP will be connected
static unsigned int strip_0_port_0;

//  SSP handles
CellAANHandle stereo_player;

//Param 1 = Channels, Param 2 = unused, must be 0
static CellSSPlayerConfig sspConfigStereo = {2, 0};

//cell\sample_data\sound\waveform\m_stereo.wav
#define STEREO_DATA "/app_home/m_stereo.wav"

int SystemAudioUtilitySet6ch(void) {
  int err, num, ch_pcm, ch_ac3,ch_dts ;
  CellAudioOutConfiguration aConfig;
  CellAudioOutState a_state;

  // Find out how many audio channels are available --------

  // Linear PCM, is always available with at least 2 channels.
  ch_pcm = cellAudioOutGetSoundAvailability(
      CELL_AUDIO_OUT_PRIMARY,
      CELL_AUDIO_OUT_CODING_TYPE_LPCM,
      CELL_AUDIO_OUT_FS_48KHZ, 0
    );
```

```
  // How many Dolby AC−3 channels available?
  ch_ac3 = cellAudioOutGetSoundAvailability(
      CELL_AUDIO_OUT_PRIMARY,
      CELL_AUDIO_OUT_CODING_TYPE_AC3,
      CELL_AUDIO_OUT_FS_48KHZ, 0
    );

  // How many Dolby DTS channels available?
  ch_dts = cellAudioOutGetSoundAvailability(
      CELL_AUDIO_OUT_PRIMARY,
      CELL_AUDIO_OUT_CODING_TYPE_DTS,
      CELL_AUDIO_OUT_FS_48KHZ, 0
    );

  // Chose an output mode −−−−−−−−−−−−−

  // The target structure must be zero cleared.
  memset(&aConfig, 0, sizeof(CellAudioOutConfiguration));

  if( ch_pcm >= 6 ) {
    // If 6ch of linear PCM is available, we use this.
    aConfig.channel = 6;
    aConfig.encoder=CELL_AUDIO_OUT_CODING_TYPE_LPCM;
    printf("Audio Encoder: PCM, Channels: 6\n");
  }
  else if( ch_ac3 >= 6 ) {
    // else if AC−3 is available, we use this.
    aConfig.channel = 6;
    aConfig.encoder = CELL_AUDIO_OUT_CODING_TYPE_AC3;
    printf("Audio Encoder: AC−3, Channels: 6\n");
  }
  else if( ch_dts >= 6 ) {
    // else if DTS is available, we use this.
    aConfig.channel = 6;
    aConfig.encoder = CELL_AUDIO_OUT_CODING_TYPE_DTS;
    printf("Audio Encoder: DTS, Channels: 6\n");
  }
  else {
    // else if neither is available, use 2ch of linear PCM.
    aConfig.channel   = 2;
    aConfig.encoder=CELL_AUDIO_OUT_CODING_TYPE_LPCM;
    printf("Encoder: PCM, Channels: 2, Downmixed from 6\n");
    // We must then downmix the multiple channels to stereo.
    aConfig.downMixer =
        CELL_AUDIO_OUT_DOWNMIXER_TYPE_A;
  }

  // Apply the selected audio settings −−−−−−−−−−−−−

  // This may take some time if the audio system is busy
  // Keep looping until it works or there is an error.
  for(num=0; num< 400; num++ ) {
    err = cellAudioOutConfigure(
        CELL_AUDIO_OUT_PRIMARY, &aConfig, NULL, 0
      );
    if( err == CELL_OK ) {
      break; //Successs!
    }
    else if(err!=CELL_AUDIO_IN_ERROR_CONDITION_BUSY)
    {
      break;
    }
    sys_timer_usleep(5000);
  }

  // Check to see if the audio system is now ready. Loop like before.
  for(num=0; num< 400; num++ ) {
    err = cellAudioOutGetState(
        CELL_AUDIO_OUT_PRIMARY, 0, &a_state
      );
    if( err == CELL_OK ) {
      if( a_state.state ==
          CELL_AUDIO_OUT_OUTPUT_STATE_ENABLED )
      {
        // Audio is enabled and ready
        break;
      }
    }
    else if(err!=CELL_AUDIO_OUT_ERROR_CONDITION_BUSY)
    {
      break;
    }
    sys_timer_usleep(5000);
  }
  return err;
}

int main(void) {
  int err;

  //Load the Audio system module
  err = cellSysmoduleLoadModule(CELL_SYSMODULE_AUDIO);

  // Configure the audio output mode
  err = SystemAudioUtilitySet6ch();

  // Initialize audio system
  err = cellAudioInit();

  // Create a mixer object
  err = cellSurMixerCreate(&g_SurMixerConfigS0);

  // Get handle of Surround Mixer
  err = cellSurMixerGetAANHandle(&mixerHandle);

  // Get port number of Channel Strip
  err = cellSurMixerChStripGetAANPortNo(
      &strip_0_port_0,CELL_SURMIXER_CHSTRIP_TYPE2A,0
    );

  // Generate Simple Sound Player
  err = cellSSPlayerCreate(&stereo_player, &sspConfigStereo);

  // Connect stereo_player to No. 0 of 2ch Channel Strip
  err = cellAANConnect(
      mixerHandle, strip_0_port_0, stereo_player, 0
    );

  // Read the Wav File
  err = Wave::readWavfile(STEREO_DATA);

  // Setup playback Parameters
  CellSSPlayerWaveParam waveInfo;
  CellSSPlayerCommonParam loopIno;
  CellSSPlayerRuntimeInfo playbackInfo;

  waveInfo.addr       = (void*)Wave::wavetop;
  waveInfo.samples     = Wave::waveByteSize / 4;
  waveInfo.loopStartOffset = 1;
  waveInfo.startOffset   = 1;
  loopIno.loopMode = CELL_SSPLAYER_LOOP_ON;
  loopIno.attackMode = 0;
  playbackInfo.level = 1.0;
  playbackInfo.speed = 1.0;

  //Send the Parameters to the Sound Player
  err = cellSSPlayerSetWave(stereo_player, &waveInfo, &loopIno);

  //Starts The mixer
  err = cellSurMixerStart();

  // Start Playback!
  printf("Starting SSP playback\n");
  cellSSPlayerPlay(stereo_player, &playbackInfo);

  //Continuously Do Nothing
  while (1) {
    sys_timer_usleep(80000);
  }

  printf("Exiting\n");
  return 0;
}
```

## 6. Conclusion

Using the Simple Sound Player to play Wav files can be done with very little additional code on top of the foundation audio code. If you need to play more than one sound, you will need a system to either control multiple SSP instances or to swap out the audio data.

## Recommended Reading

Programming the Cell Processor: For Games, Graphics, and Computation, Matthew Scarpino
ISBN: 978-0136008866
Waveform Audio File Format, Wikipedia
en.wikipedia.org/wiki/WAV

## References

[1] Edinburgh Napier Game Technology Website. www.napier.ac.uk/games/. *Accessed: Feb 2014*, 2014. 1