Audio in PNaCl application

Published 2014-10-27 | (Compatible with SDK 4.5,5.0,5.1 and 2013,2014 models)

> This document shows how to use PNaCl Audio interface in SmartTV application

Contents

This tutorial provides information on how to create a PNaCl application that uses **pp::Audio** and **pp::URLLoader** resources. The application can be tested with Samsung Smart TV Emulator 4.5 or directly on Samsung Smart TV with Native Client. You can also test it in the Google Chrome browser.

Prerequisites

To create a PNaCl application you'll need NaCl SDK and a text editor. Basic tutorial on creating PNaCl applications can be found in How to create sample PNaCl application.

To run the application this tutorial describes, please download the **tutorial application source code** and extract it into the Samsung SmartTV SDK application folder.

You will also need an audio device (i.e. headphones or speaker) and a .wav file.

Introduction

The **pp::Audio** interface in PNaCl enables usage of sounds in your applications.

There are 3 possible audio sources:
an audio file (for example in WAVE format)
a sound generated inside of application (i.e. sine wave)
a sound recorded into the memory by another part of application

All 3 types of sources can be joined to create more complex sounds.

This tutorial shows how to create a PNaCl application that reads data from a WAVE file and plays it on an audio output device, so that user can hear it. Also application manages mixing and playing simultaneously sounds from different files.

Interfaces description

Reading from file and playing sounds are both not trivial operations. They both require more than one type of resource. Therefore, we will describe them briefly.

The pp::Audio interface

The **pp::Audio** provides a low-level audio stream handling. When resource is created, it gets **PPB_Audio_Callback** callback function, that gets called every time browser needs more data to play. Playback can be stopped or started.

More information here: **coding Audio**.

The pp::URLLoader interface

When running PNaCl module, it does not have access to file system. If you need any additional assets (such as files) from specific location, you have to use **pp::URLLoader** interface.

More information here: **coding URL Loading**.

Using tutorial application

1. When application initializes, three .wav files with audio data are immediately loaded.
2. When files are loaded, corresponding Play buttons will be enabled.

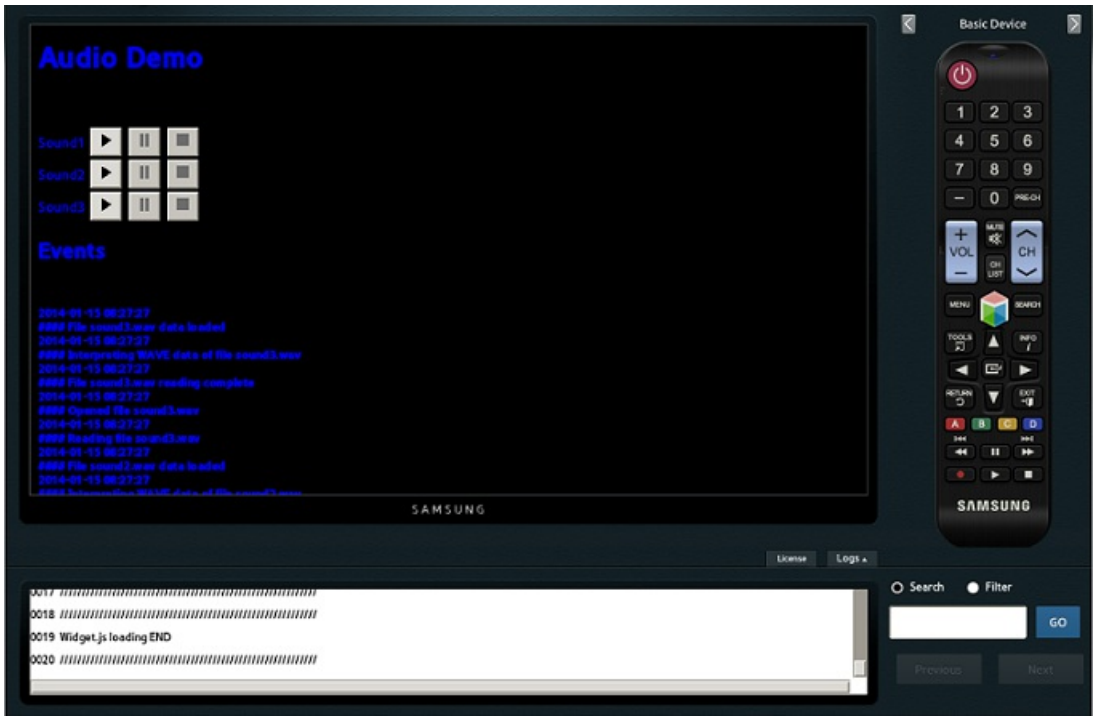

Figure 1: Application with all files loaded

3. Clicking one of Play buttons will start playback of sound from equivalent .wav file. Sound will be played once.

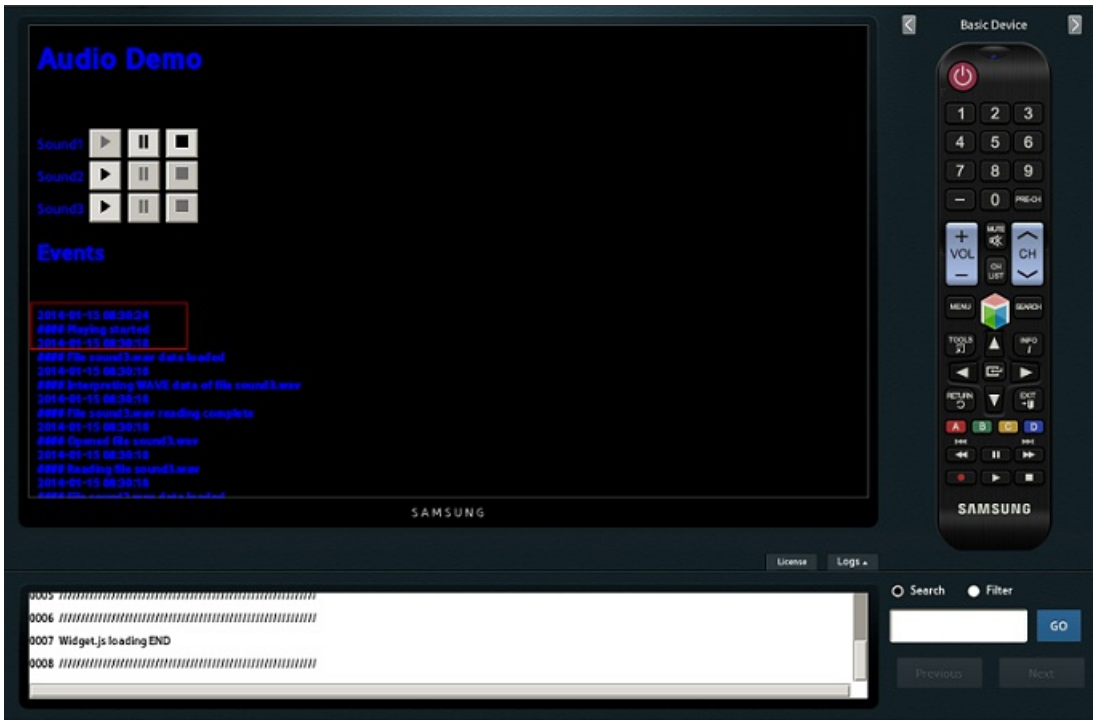

Figure 2: Application playing sound

4. Playback is stopped, when Pause or Stop button is pressed. Play button will be enabled again. Pressing it will resume playing or start from the beginning.

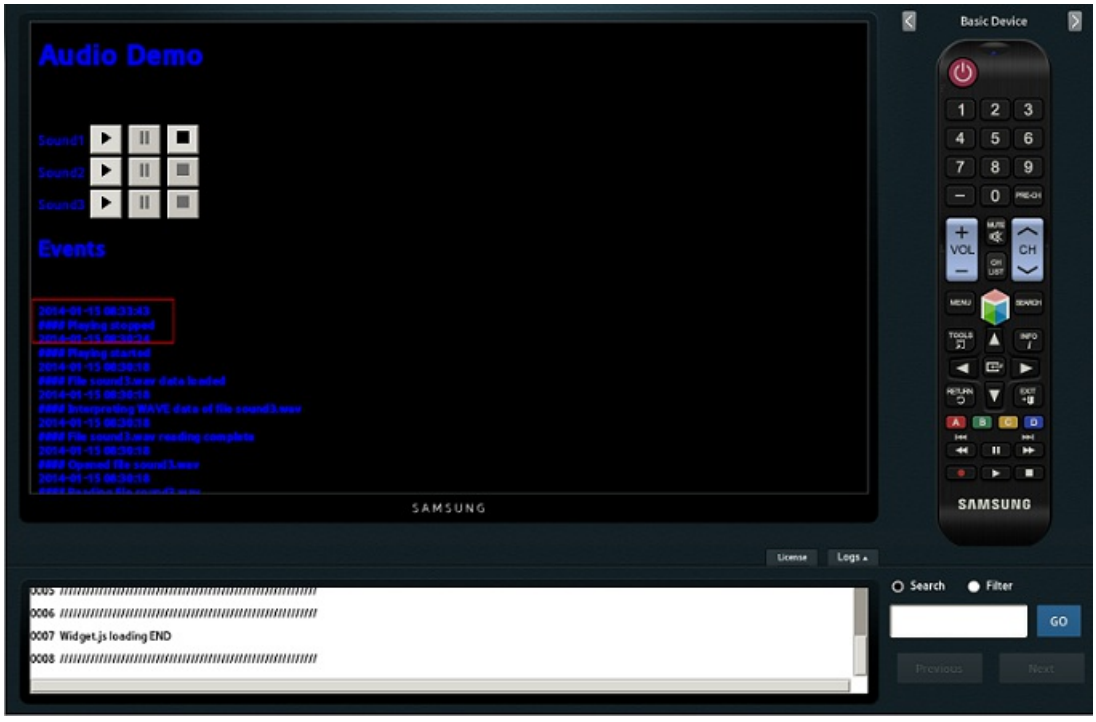

Figure 3: Playing was paused

5. Playing multiple sounds simultaneously.

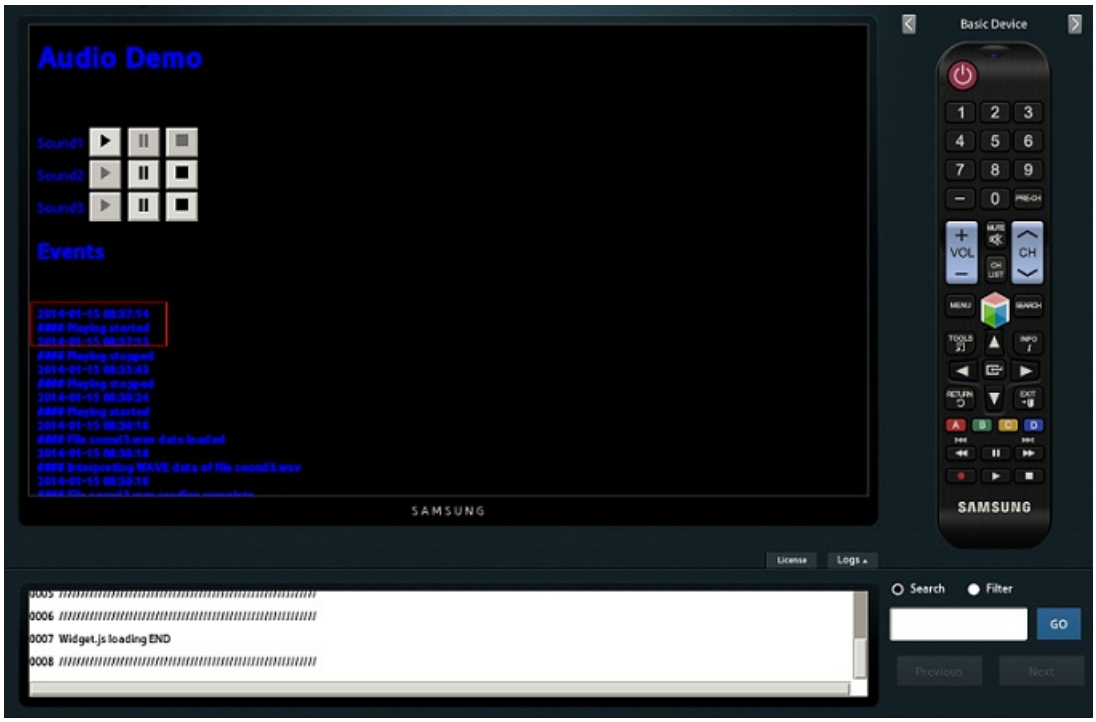

Figure 4: Two sounds are being played simultaneously. Third sound is stopped.

Problems with loading files

1. An error occurred: **Could not read file** and Play button is not enabled

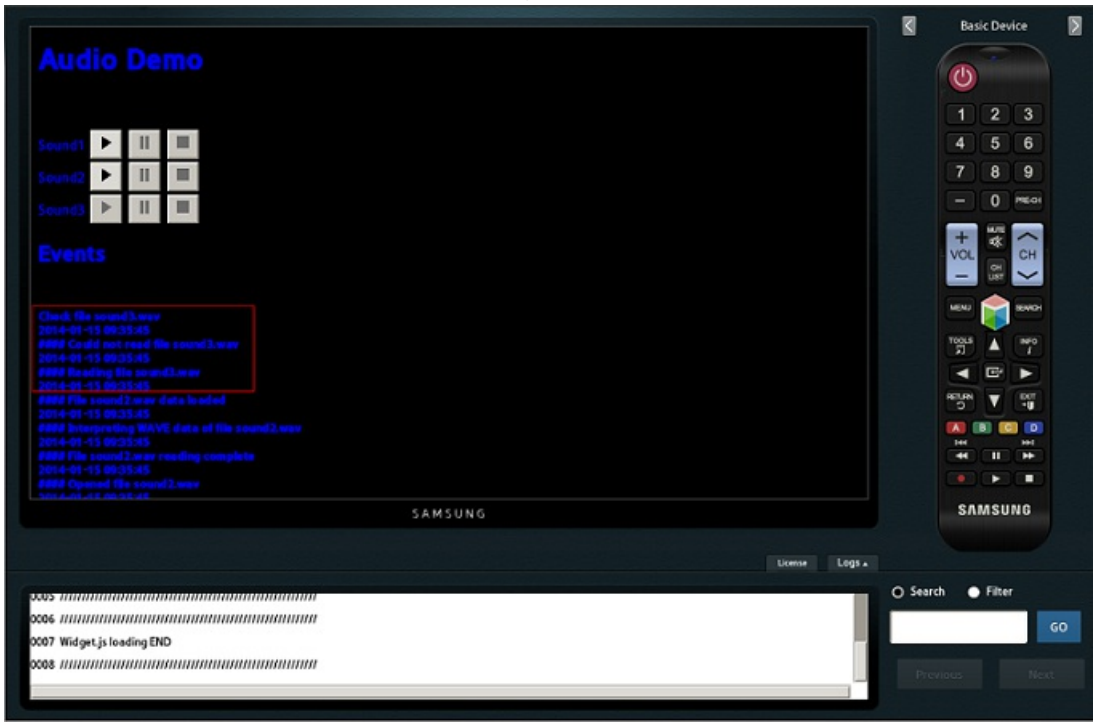

Figure 5: Could not read file error

The file doesn't exist or can't be opened. Check if the file is placed in sounds folder in your widget directory and is granted permission to be read. Then refresh application to reload it.

2. An error occurred: **Error in reading file** and Play button is not enabled

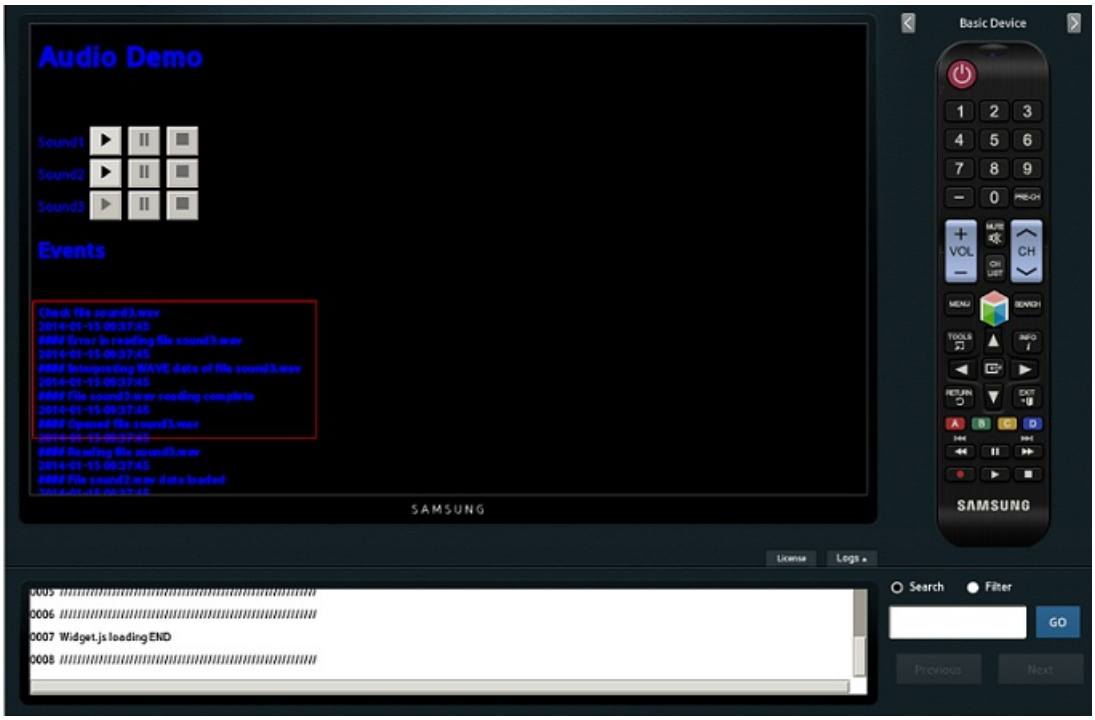

Figure 6: Error in reading file error

The file header is malformed and WAVE file couldn't be interpreted. Correct it and refresh application to reload it.

Creating your own application

Loading files

> Note
>
> The **pp::URLLoader** serves mainly for loading remote resources, but we can use it also for loading assets from widget directory. When doing this, you need to directly specify needed file path from your widget directory:
>
> ```
> // set file to download
> url_ = "./sounds/" + varMessage.AsString() + ".wav";
> ```

Following figure shows how browser and PNaCl module communicate when loading requested file:

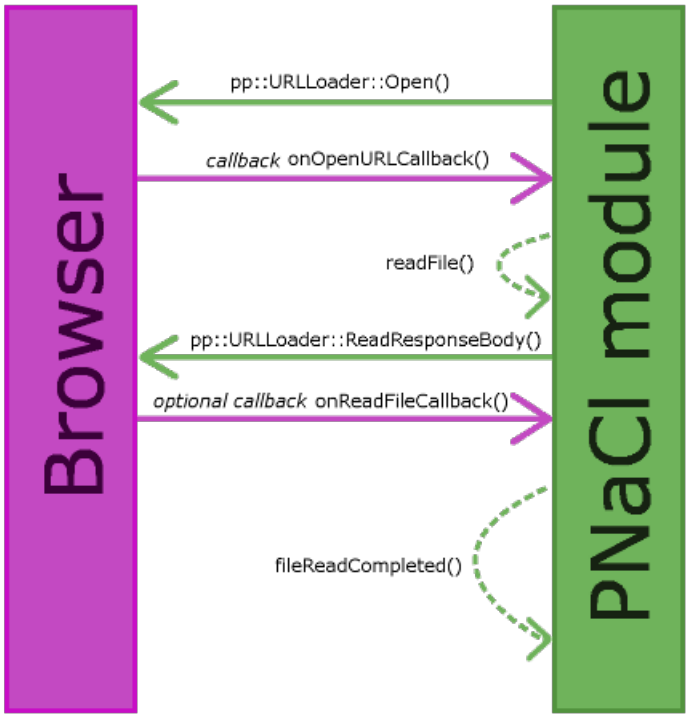

Figure 7: Communication between browser and PNaCl module during loading file

As shown on Figure 7: Communication between browser and PNaCl module during loading file we need to implement two callbacks:

first will start reading procedure,

second, called when data was read, should be created as optional, because when data is available (here we don't need to download it from remote point, so it is available) ReadResponseBody() may return read data synchronously.

When full file was read or an error occurred, fileReadCompleted() function is called. This function starts WAVE interpreting procedure.

Reading WAVE header

The Microsoft's RIFF specification contains the WAVE file format. File starts with a header followed by a sequence of samples formatted according to data in the header. The standard WAVE file format is as follows:

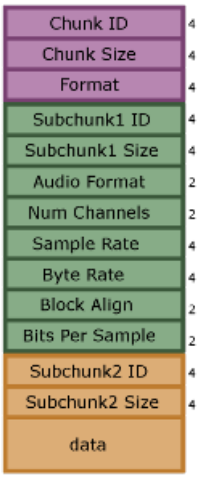

Figure 8: WAVE format

From this format we will extract a header with fields of appropriate size:

```
struct FileHeader
{
  char chunkID[4];
  int32_t chunkSize;
  char format[4];

  char subchunk1ID[4];
  int32_t subchunk1Size;
  int16_t audioFormat;
  int16_t numChannels;
  int32_t sampleRate;
  int32_t byteRate;
  int16_t blockAlign;
  int16_t bitsPerSample;

  char subchunk2ID[4];
  int32_t subchunk2Size;
};
```

Now we should check whether the header in file is correct.

1. Check if chunk names and format have appropriate value for WAVE file.
2. Check if file is not compressed.
3. Check if bits per sample rate is appropriate.
4. Check if chunk sizes are correct.

> **Important**
>
> Remember that chunk size value does not include chunk ID and size fields.

5. Check if number of channels is lesser or equals 2 - PNaCl supports only stereo sound.

```
if (std::string(header_->chunkID, 4) != "RIFF" ||
  std::string(header_->format, 4) != "WAVE" ||
  std::string(header_->subchunk1ID, 4) != "fmt " ||
  std::string(header_->subchunk2ID, 4) != "data" ||
  header_->audioFormat != 1 || // without compression
  header_->bitsPerSample != 16 ||
  header_->chunkSize != 4 + (8 + header_->subchunk1Size) + (8 + header_->subchunk2Size)||
  (header_->numChannels != 1 && header_->numChannels != 2)) // supporting only mono and stereo sound
```

Playing sounds

Once sound is being played, browser continually sends request to fill audio buffer. PNaCl supports stereo sound. This means that we have to provide two samples (left and right) for every sample to play.

Mixing sounds is not supported by **pp::Audio** interface. To achieve multiple sounds plying simultaneously mixing technique has to be implemented by application. This tutorial provides simple mixer that sums up samples from all playing sounds. Volumes of the sounds are normalized according to number of playing sounds.

For mono sound we just rewrite one sample many times to output buffer.

```
// for mono sound each sample is passed to both channels
audioDemoInstance->safeAdd(buff[i*2], (int16_t)instance.sampleData[instance.sampleDataOffset] * volume);
audioDemoInstance->safeAdd(buff[i*2+1], (int16_t)instance.sampleData[instance.sampleDataOffset] * volume);
instance.sampleDataOffset++;
```

For stereo sound each sample is used once.

```
// for stereo sound samples are written successively to all channels
audioDemoInstance->safeAdd(buff[2*i], (int16_t)instance.sampleData[instance.sampleDataOffset] * volume);
instance.sampleDataOffset++;
audioDemoInstance->safeAdd(buff[2*i+1], (int16_t)instance.sampleData[instance.sampleDataOffset] * volume);
instance.sampleDataOffset++;
```

> Important
>
> Always check before writing to buffer if its size is enough to write as much samples as required.

Thread and real-time issues

As audio callback function is called from background application thread. This brings several problems:

1. **Data access**

   Data can be accessed from both audio and main application threads. However, locks should not be used, as attempting to acquire a lock may result in swapping out the thread and audio dropouts.
2. **PPAPI and CRT (C Run-Time) library calls**

   Using some functions form PPAPI or CRT library, such as malloc, gettimeofday, mutex, critical sections also swaps out the callback function and shouldn't be used inside of it.
3. **Long computations**

   Another case when audio dropouts may occur is when callback computation time does not meet real-time conditions.

Summing up, you should program the callback function very carefully. Calling some functions or doing much time-consuming computations may result in hard to track down and debug audio dropouts.

> Warning
>
> StartPlayback() and StopPlayback() are asynchronous RPCs. That means, the playback may not be started or stopped immediately after calling these functions. If you need it to be precise with another application actions, you have to synchronize it manually.