Professional Audio

Programming Guide

Version 1.0

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

Table of Contents

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

1. Overview

Professional Audio allows you to create virtual instrument applications with Android. You can connect to and share audio devices and synchronize low-latency shared devices.

Professional Audio improves the environment in which virtual instruments are created by adding high-performance audio processing logic. You can use Professional Audio to create applications without background knowledge in hardware and high-performance drivers. You need not worry about connecting devices between applications. Using the provided modules and a USB MIDI driver, you can create virtual instrument applications with ease.

You can use the Professional Audio package to:

- Create professional musical instrument applications using JACK Audio Connection Kit features.
- Send MIDI data, control audio/MIDI ports, access/use added plug-in information, and synchronize virtual instruments.
- Create new sound modules and process high-speed audio signals.
- Include plug-ins for acoustic piano, steel guitar and a standard drum kit.

1.1. Basic Knowledge

Professional Audio is built on the following frameworks:

ALSA

The Advanced Linux Sound Architecture (ALSA) provides audio and MIDI functionality for the Linux operating system. ALSA offers the following features:

- Support for all audio interface types, from consumer sound cards to professional multichannel audio interfaces.
- Fully modularized sound drivers.
- SMP and thread-safe design. For more information, click this link.
- Library (alsa-lib) to simplify application programming and to provide higher level functionality.
- Support for the older Open Sound System (OSS) API, providing binary compatibility for most OSS programs.

JACK

JACK is a system for handling real-time, low-latency audio and MIDI. It runs on GNU/Linux, Solaris, FreeBSD, OS X, and Windows and can be ported to other POSIX-conformant platforms. It can connect a number of different applications to an audio device and allow them to share audio. Its clients can run in their own processes as normal applications, or they can run within the JACK server as a plug-in. JACK also supports distributing audio processing across a network, both in fast and reliable LANs and in slower, less reliable WANs.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

JACK was designed from the ground up for professional audio work, and its design focuses on two key areas: synchronous execution of all clients and low-latency operation.

APAService

APAService provides an environment that enables JACK to run on Android and serves as a connection between APAClient and Android applications.

Shared Memory Service

Shared Memory Service provides a shared memory for facilitating fast and continual data communications between processes and for enabling audio, MIDI, and control data sharing between JACK daemons and JACK clients.

1.2. Architecture

The following figure shows the Professional Audio architecture.

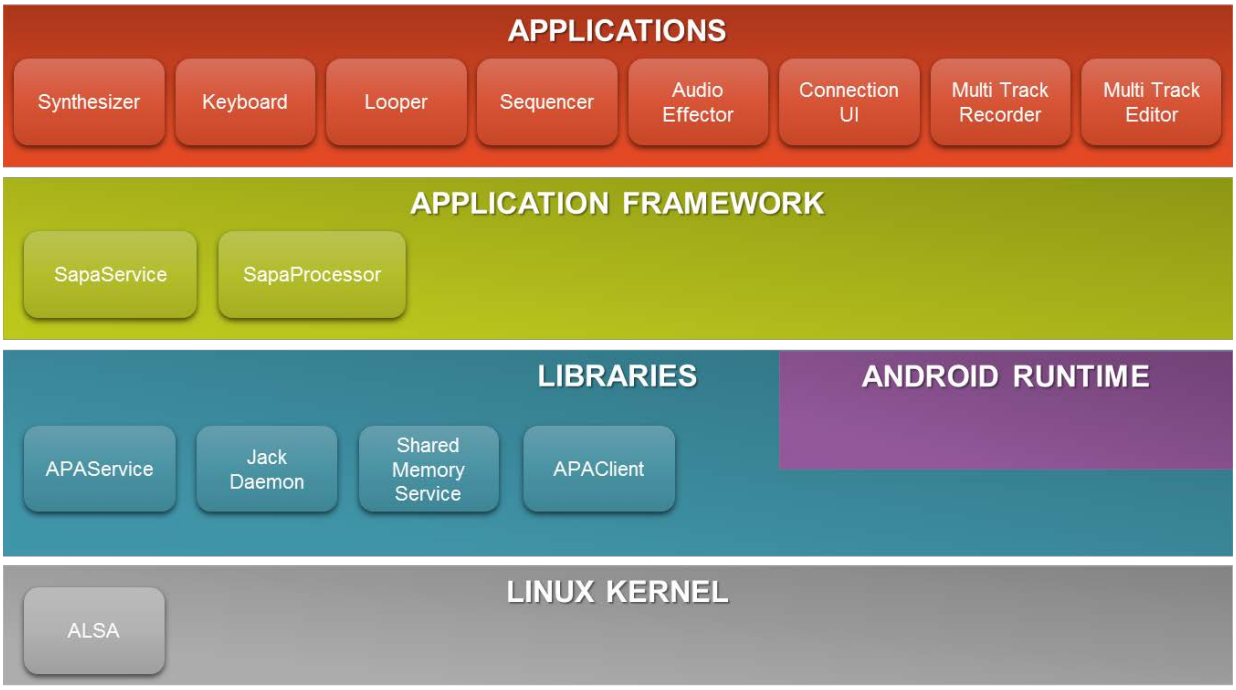

Figure 1: Professional Audio architecture

The architecture consists of:

- **Applications:** One or more applications that are integrated with the Professional Audio package. By default, Professional Audio does not supply these kinds of applications.
- **SapaService:** Component for connecting with JACK and for managing ports, processors, and connections.
- **SapaProcessor:** Component for managing processors that handle audio stream or MIDI data.
- **APAService and APAClient:** Components for controlling JACK and making it available on Android .
- **Shared Memory Service**: Component for facilitating data sharing between processing modules.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

- **ALSA:** Component that provides audio and MIDI functionality for the Professional Audio package.

1.2.1. Distributing Your Application

There are three ways of distributing your Professional Audio applications:

- **Type1:** Include your own sound processing modules within your APK.
- **Type2:** Rely on plug-in processing modules. This approach is useful when you want to concentrate on UI development only. For example, when you use the SapaPiano plug-in, you only need to create and distribute your piano UI.
- **Type3:** Include processing modules only. You can develop and distribute your own sound processing modules without UIs in APKs.

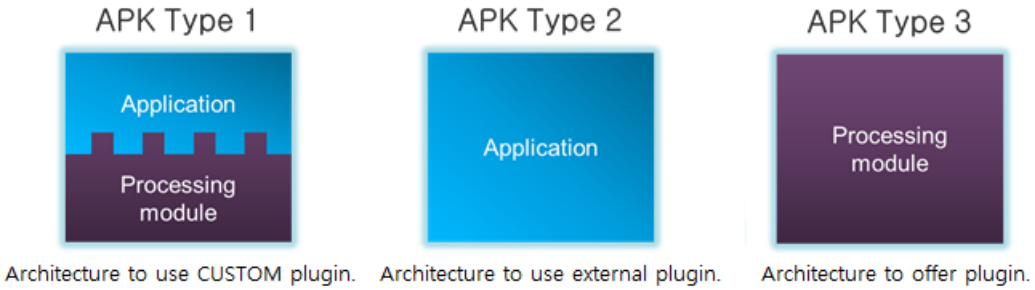

Figure 2: Application types

The following figure shows the data flow when 2 Type 1 applications are running:

The applications are designed to operate with 2 types of data: audio data and control data. Audio data flows according to the connection information in the port that is connected through the JACK daemon. The JACK daemon is responsible for handling the audio data.

APAService connects and disconnects ports between applications and communicates with applications and processing modules.

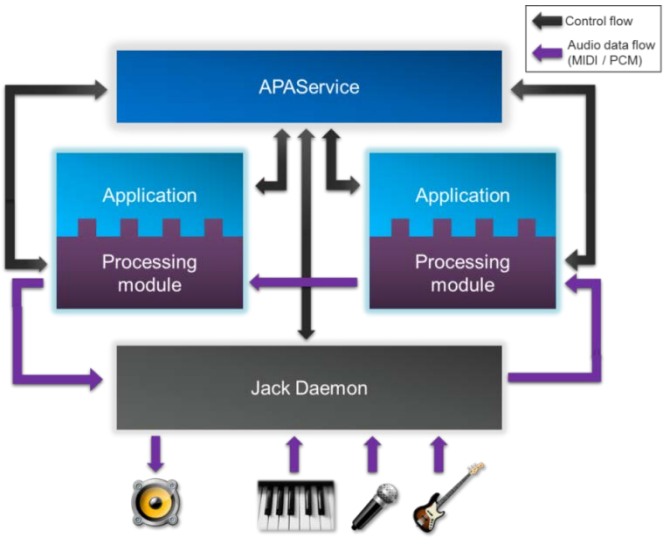

Figure 3: Data flow between 2 type1 applications

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

You can run a processing module by activating the corresponding SapaProcessor object. When you register a SapaProcessor object with SapaService, it creates a new process for the processing module.

The following figure shows an application and a processing module being run at the same time.

Applications are executed in the same manner as an android application process forked by a zygote. They communicate with ApaService through SapaService. When you create and register a SapaProcessor object with SapaService, it creates a new process. The new process contains the APAClient and your processing module. You can use SapaProcessor to communicate with the process that loads the processing module.

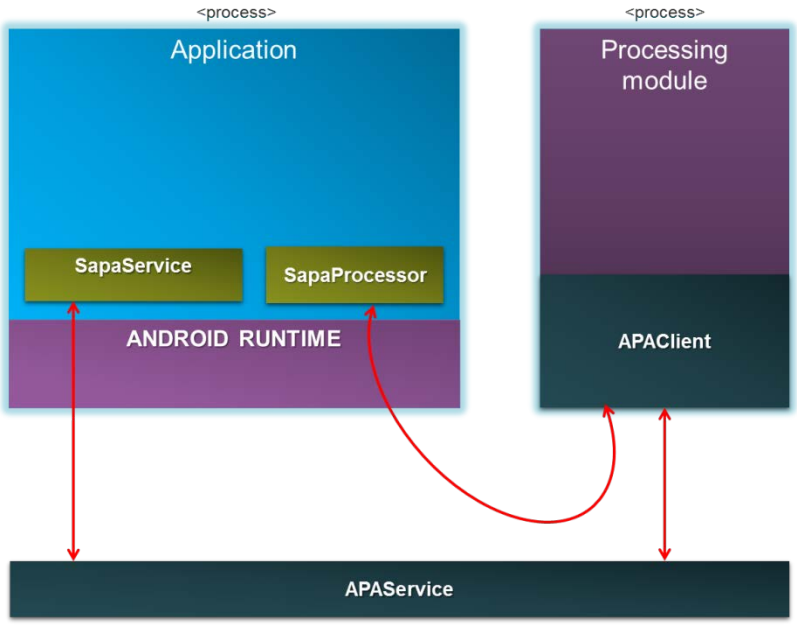

Figure 4: Processes for application and processing module

1.2.2. Ports

JACK ports connect processing modules. To have a speaker produce a sound for example, the sound data needs to be transferred to the port linked to the speaker. There are 2 types of ports: input ports, for sound inputs, and output ports, for sound outputs.

Professional Audio provides the following ports by default. The ports available also depend on the device:

Input Port	Description
system:capture_1	Connected to phone microphones. • If a stereo microphone is connected, this port is connected to the left microphone. • This port is used to connect to mono microphones. • If an earphone is connected, this port is connected to the earphone microphone.
system:capture_2	Connected to phone microphones. • If a stereo microphone is connected, this port is connected to the right microphone. • If a mono microphone is connected, this port gets the same input as

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

Input Port	Description
	capture_1. • If an earphone is connected, this port gets the same input as capture_1.
loopback:capture_1	A virtual port that cannot be connected to any devices. This port is directly connected to loopback:playback_1 and can switch output sounds to input sounds.
loopback:capture_2	A virtual port that cannot be connected to any devices. This port is directly connected to loopback:playback_2 and can switch output sounds to input sounds.

Output Port	Description
system:playback_1	Connected to phone speakers. • If stereo speakers are available, this port is connected to the left speaker. • By default, this port is connected to mono speakers. • If an earphone is connected, this port is connected to the left earphone speaker.
system:playback_2	Connected to phone speakers. • If stereo speakers are available, this port is connected to the right speaker. • If a mono speaker is connected, this port is mixed with playback_1. • If an earphone is connected, this port is connected to the right earphone speaker.
loopback:playback_1	A virtual port that cannot be connected to any devices. This port is connected to produce sounds with Loopback:capture_1.
loopback:playback_2	A virtual port that cannot be connected to any devices. This port is connected to produce sounds with Loopback:capture_2.

You can call the `SapaService.getAllPort()` method to get all the available ports.

Port connections are used when ports are connected; for example, a connection between system:capture_1 and system:playback_1. You can call the `SapaService.getAllConnection()` method to get all the current port connections.

1.2.3. Plug-in Structure

Type 2 applications use plug-ins distributed in other APKs, while Type 3 applications provide their plug-ins in APKs for use by other applications. When the plug-in APKs are registered correctly with the device, the Professional Audio package can find the plug-ins and make them available to your applications.

Call the `SapaService.getPluginList()` method to get the list of plug-ins installed on the device. Call the `SapaService.PluginInfo()` method to get the following information for each plug-in:

- **Name:** The name of the plug-in.
- **Version:** The plug-in version number.
- **Version Name:** The plug-in version name as a string.
- **Package Name:** The plug-in package name. When plug-in names overlap, use the package names to distinguish them.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

In order for the PluginInfo class methods to return the correct values, the APKs that provide the plug-ins must be packaged properly.

For information on creating Type 2 applications that use already installed plug-ins, see Hello Sapa.

For information on distributing processing modules, see Distributing Type3 Applications.

1.3. Class Diagram

The following figure shows the Professional Audio classes and interfaces that you can use in your application.

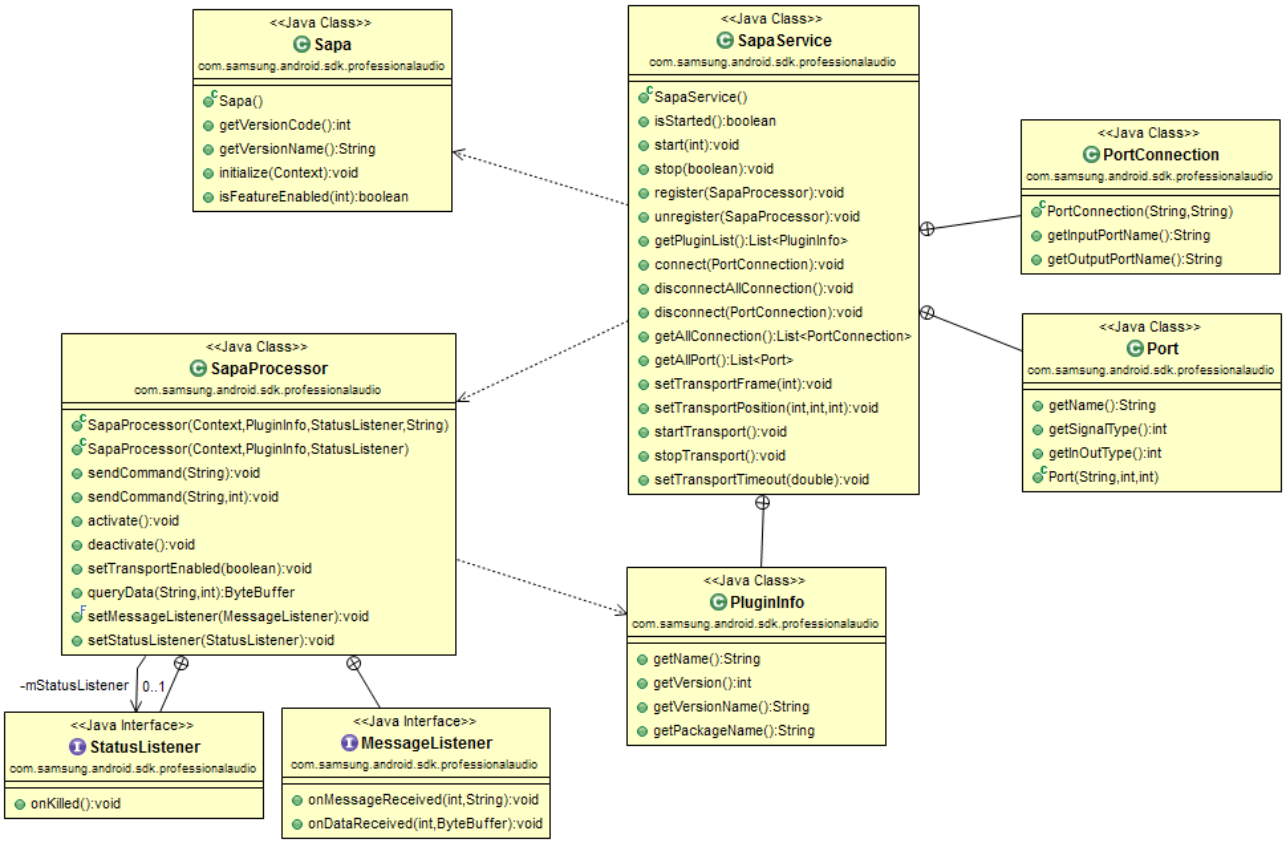

Figure 5: Professional Audio classes and interfaces

The Professional Audio classes and interfaces include:

- **Sapa:** Initializes the Professional Audio package.
- **SapaService:** Registers SapaProcessor to make it available and manages links between SapaProcessors.
- **SapaProcessor:** Creates and manages processing modules, and handles sound processing.
- **StatusListener:** Listens for processing module status change events.
- **MessageListener:** Listens for messages or data from processing modules.
- **PluginInfo:** Contains plug-in information.
- **Port:** Provides port functionality.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

- **PortConnection:** Provides port connection functionality.

1.4. Supported Platforms

- Samsung Galaxy S series and Samsung Galaxy Note series support Professional Audio.
- Android 4.1 (JellyBean API 16) or above support Professional Audio.

1.5. Supported Features

Professional Audio supports the following features:

- An environment where a range of virtual music instruments or devices can run.
- An environment that enables various devices to easily connect to one another.

1.6. Components

- Components:
    - professionalaudio-v1.0.0.jar
    - sdk-v1.0.0.jar
- Imported packages:
    - com.samsung.android.sdk.professionalaudio

1.7. Installing the Package for Eclipse

To install Professional Audio for Eclipse:

1. Add the following files and folder to the libs folder in Eclipse:

- apa-javadoc folder and its contents
- professionalaudio-v1.0.0.jar
- professionalaudio-v1.0.0.jar.properties
- sdk-v1.0.0.jar

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

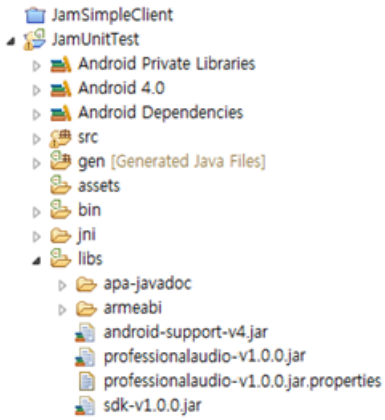

Figure 6: libs folder in Eclipse

For more information on additional actions required for developing Type 1 and Type 3 applications, see sections 4.2 and 4.3.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

2. Hello Sapa

Hello Sapa is a sample Type 2 program that uses a simple plug-in to generate a sine wave sound. This means it provides a UI that relies upon a processor plug-in that must be present on the device. To run Hello Sapa, install the SapaSimplePlugin APK.

- Project name: SapaPluginSimpleClient
- Package name: com.samsung.android.sdk.professionalaudio.sample.pluginsimpleclient
- Activity name: SapaPluginSimpleClient

Hello Sapa implements the following features:

1. Initializes the Professional Audio package.

```
try {
       Sapa sapa = new Sapa();
       sapa.initialize(this);
       mService = new SapaService();
       mService.start(SapaService.START_PARAM_DEFAULT_LATENCY);
} catch (SsdkUnsupportedException e) {
        Toast.makeText(this, "Professional audio package not supported",
        Toast.LENGTH_LONG).show();
       e.printStackTrace();
       finish();
       return;
} catch (InstantiationException e) {
        Toast.makeText(this, "Failed to instantiate SapaService",
        Toast.LENGTH_LONG).show();
       e.printStackTrace();
       finish();
       return;
}
```

2. Loads the plug-in.

```
List<SapaService.PluginInfo> pluginList = mService.getPluginList();
Iterator<SapaService.PluginInfo> iter = pluginList.iterator();
while (iter.hasNext()) {
    SapaService.PluginInfo info = iter.next();
    if (info.getName().contentEquals("SapaSimplePlugin") == true) {
        // load SapaSimplePlugin
        try {
            mProcessor = new SapaProcessor(v.getContext(), info, null);
            mService.register(mProcessor);
            ((TextView) findViewById(R.id.load_status_text)).setText("Loading SUCCESS
");
            ((Button) findViewById(R.id.activate_button)).setEnabled(true);
            ((Button) findViewById(R.id.load_button)).setEnabled(false);
            return;
       } catch (InstantiationException e) {
             Toast.makeText(v.getContext(), "Failed to register SapaProcessor",
                 Toast.LENGTH_LONG).show();
                     ((TextView) findViewById(R.id.load_status_text)).setText("Loading
FAIL ");
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

```
                    e.printStackTrace();
                }
                break;
        }
}
```

3. Activates the SapaProcessor instance.

```
if (mProcessor != null) {
        mProcessor.activate();
}
```

The complete code of the sample application is given below.

```
package com.samsung.android.sdk.professionalaudio.sample.pluginsimpleclient;

import java.util.Iterator;
import java.util.List;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

import com.samsung.android.sdk.SsdkUnsupportedException;
import com.samsung.android.sdk.professionalaudio.Sapa;
import com.samsung.android.sdk.professionalaudio.SapaProcessor;
import com.samsung.android.sdk.professionalaudio.SapaService;

public class SapaPluginSimpleClient extends Activity {

        private SapaService mService;
        private SapaProcessor mProcessor;

        @Override
        protected void onDestroy() {

                super.onDestroy();
                mProcessor.deactivate();
                mService.unregister(mProcessor);
                mService.stop(false);
        }

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_sapa_plugin_simple_client);

                try {
                        Sapa sapa = new Sapa();
                        sapa.initialize(this);
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

```
                    mService = new SapaService();
                    mService.start(SapaService.START_PARAM_DEFAULT_LATENCY);

             } catch (SsdkUnsupportedException e) {
                    Toast.makeText(this, "Professional Audio package not supported",
Toast.LENGTH_LONG).show();
                    e.printStackTrace();
                    finish();
                    return;
             } catch (InstantiationException e) {
                    Toast.makeText(this, "Failed to instantiate SapaService",
Toast.LENGTH_LONG).show();
                    e.printStackTrace();
                    finish();
                    return;
             }

             ((Button) findViewById(R.id.load_button)).setEnabled(true);

             ((Button) findViewById(R.id.load_button)).setOnClickListener(new
OnClickListener() {
                    @Override
                    public void onClick(View v) {
                           ((TextView)
findViewById(R.id.load_status_text)).setText("Loading... ");
                           List<SapaService.PluginInfo> pluginList =
mService.getPluginList();
                           Iterator<SapaService.PluginInfo> iter =
pluginList.iterator();
                           while (iter.hasNext()) {
                                  SapaService.PluginInfo info = iter.next();
                                  if (info.getName().contentEquals("SapaSimplePlugin")
== true) {
                                         // load SapaSimplePlugin
                                         try {
                                                mProcessor = new
SapaProcessor(v.getContext(), info, null);
                                                mService.register(mProcessor);
                                                ((TextView)
findViewById(R.id.load_status_text)).setText("Loading SUCCESS ");
                                                ((Button)
findViewById(R.id.activate_button)).setEnabled(true);
                                                ((Button)
findViewById(R.id.load_button)).setEnabled(false);
                                                return;
                                         } catch (InstantiationException e) {
                                                Toast.makeText(v.getContext(), "Failed
to register SapaProcessor", Toast.LENGTH_LONG).show();
                                                ((TextView)
findViewById(R.id.load_status_text)).setText("Loading FAIL ");
                                                e.printStackTrace();
                                         }
                                         break;
                                  }
                           }
                           ((TextView)
findViewById(R.id.load_status_text)).setText("Loading FAIL ");
                    }
             });
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

```
            ((Button) findViewById(R.id.activate_button)).setEnabled(false);
            ((Button) findViewById(R.id.activate_button)).setOnClickListener(new
OnClickListener() {

                  @Override
                  public void onClick(View v) {
                        if (mProcessor != null) {
                              mProcessor.activate();
                        }
                  }
            });
      }

      @Override
      public boolean onCreateOptionsMenu(Menu menu) {
            // Inflate the menu; this adds items to the action bar if it is present.
            getMenuInflater().inflate(R.menu.sapa_plugin_simple_client, menu);
            return true;
      }

}
```

For more information, see SapaPluginSimpleClient.zip.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

3. Using the Sapa Class

You need to initialize a Sapa before you can use it. Samsung Mobile SDK provides a base class with an initialize() method for each package.

The Sapa can run only on Samsung Smart Devices. Some Samsung Smart Device models do not support some *of the* packages.

You can use a `initialize()` method to initialize it and also to check if the device supports the Sapa. If the device does not support the Sapa, the method throws an SsdkUnsupportedException exception. You should handle this exception. If an SsdkUnsupportedException exception is thrown, you can check the exception type with **`SsdkUnsupportedException.getType()`**. If the device is a Samsung model that does not support the Sapa, the exception type is **SsdkUnsupportedException.DEVICE_NOT_SUPPORTED**.

The Sapa class provides the following methods:

- `initialize()` initializes Professional Audio. You need to initialize the Professional Audio package before you can use it. If the device does not support Professional Audio, SsdkUnsupportedException is thrown.
- `getVersionCode()` gets the Professional Audio version number as an integer.
- `getVersionName()` gets the Professional Audio version name as a string.
- `isFeatureEnabled()` checks if a Professional Audio package feature is available on the device.

```
Sapa sapa = new Sapa();
try {
    // Initialize the instance of Sapa
    pass.initialize(this);
} catch (SsdkUnsupportedException e) {
    // Error handling
}

int versionCode = sapa.getVersionCode();

String versionName = sapa.getVersionName();
```

For more information, see on`Create()` in SapaSimpleClientActivity.java in SapaSimpleClient.

3.1. Using the initialize() Method

The Sapa.initialize() method:

- initializes the Professional Audio package

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

- checks if the device is a Samsung device
- checks if the device supports the Professional Audio package
- checks if the Professional Audio libraries are installed on the device

```
void initialize(Context context) throws SsdkUnsupportedException
```

If the Professional Audio package fails to initialize, the initialize() method throws an SsdkUnsupportedException exception. To find out the reason for the exception, check the exception message.

3.2. Handling SsdkUnsupportedException

If an SsdkUnsupportedException exception is thrown, check the exception message using SsdkUnsupportedException.getType().

The following type of exception message is defined in the Sapa class:

- **DEVICE_NOT_SUPPORTED:** The device does not support the Professional Audio package.

3.3. Checking the Availability of Professional Audio Package Features

You can check if a Professional Audio package feature is supported on the device with the isFeatureEnabled() method. The feature types are defined in the Sapa class. Pass the feature type as a parameter when calling the isFeatureEnabled() method. The method returns a Boolean value that indicates the support for the feature on the device.

```
boolean isFeatureEnabled(int type);
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

4. Using the Professional Audio Package

4.1. Creating a Type 2 Application

The following sections provide details on creating an application that provides a UI, but relies on a plug-in on the device for processing. This example uses the sample piano processor included with the Professional Audio package.

4.1.1. Starting the Service

To start the Professional Audio package:

1. Create an instance of the Professional Audio package and initialize it by calling the `Sapa.initialize()` method. If Professional Audio is not available on the device, an SsdkUnsupportedException exception is thrown.

```
try {
	Sapa sapa = new Sapa();
	sapa.initialize(this);

} catch (SsdkUnsupportedException e) {
} catch (InstantiationException e) {
}
```

2. To start the Professional Audio service with the default latency, call the `SapaService.start()` method and pass SapaService.START_PARAM_DEFAULT_LATENCY:

```
try {
	(new SapaService()).start(SapaService.START_PARAM_DEFAULT_LATENCY);
} catch (AndroidRuntimeException e) {
}
```

4.1.2. Getting the Plug-ins

Processors for musical instruments or devices for Professional Audio are handled as plug-ins. For example, to make your musical instrument application play an ocarina sound, install the processing module plug-in for an ocarina.

To get the list of plug-ins available on your device, call `SapaService.getPluginList()`.

To use the SapaPaino plug-in referred to in the code below, you need to install SapaFluidSynthPlugin.apk.

```
List<SapaService.PluginInfo> pluginList = (new SapaService()).getPluginList();
Iterator<SapaService.PluginInfo> iter = pluginList.iterator();
while (iter.hasNext()) {
	SapaService.PluginInfo info = iter.next();
	if (info.getName().contentEquals("SapaPiano") == true) {
		// TODO: Use info
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

```
        }
}
```

4.1.3. Making the Processor Available

To load a plug-in and make it available:

1. Create a new SapaProcessor instance for the plug-in and register it with the SapaService.

```
List<SapaService.PluginInfo> pluginList = (new SapaService()).getPluginList();
Iterator<SapaService.PluginInfo> iter = pluginList.iterator();
    while (iter.hasNext()) {
       SapaService.PluginInfo info = iter.next();
       if (info.getName().contentEquals("SapaPiano") == true) {
        try {
              mService.register(new SapaProcessor(context, info, null));
       } catch (InstantiationException e) {
       }
    }
}
```

4.1.4. Using the Processing Module

For information on how to use a processing module, see the user manual for that processing module.

To send commands to the SapaPiano processing module:

1. In the `onClick()` method for your UI, call the `SapaProcessor.sendCommand()` method and pass the note and velocity.

```
private SapaProcessor mProcessor;
public void onClick(View v) {
    // mProcesor is assumed to be registered to SapaService.
    final int noteC1 = 62;
    final int velocity = 90;
    mProcessor.sendCommand(String.format("PLAY %02x %02x", noteC1, velocity));
}
```

4.1.5. Closing the Processing Module

Close the processing module after use to release the resources.

To close the processing module:

1. Call the `SapaService.unregister()` method.

```
private SapaProcessor mProcessor;
public void onClick(View v) {
    // mProcesor is assumed to be registered to SapaService.
    mService.unregister(mProcessor);
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

```
}
```

4.1.6. Stopping the Service

Stop the Professional Audio service after use. The service uses other device resources than the processing modules.

To stop the service:

1. Call the `SapaService.stop()` method.

```
(new SapaService()).stop(false);
```

If you did not unregister the processing module, the service will not stop. Unregister all the processing modules registered with the service before calling the `SapaService.stop()` method.

4.2. Creating a Type 1 Application using the NDK

You can use the Professional Audio NDK to develop custom processing modules (“CUSTOM” plug-ins).

The following sections demonstrate how to develop an application that plays a sine wave sound, replace the appropriate sections with your own implementation.

4.2.1. Setting up the Environment

Download and install Android NDK.

1. Go to the Android NDK website and download the NDK for Windows.
2. Unzip the package to the C:\ drive. This creates a new folder called android-ndk-r9*.* in the C:\ drive.
3. To set up the environmental variables on your computer, in the window for setting environmental variables, in the **Path** textbox, add C:\android-ndk-r9.
4. Test your settings in the command line as shown below:

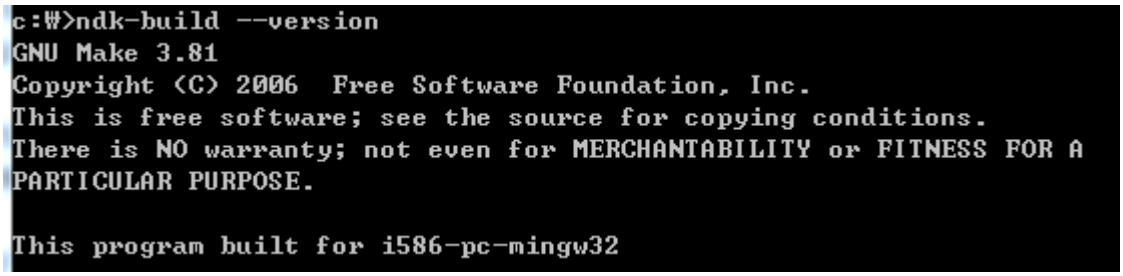

Figure 7: Checking if the installation is successful

5. Copy and paste the Professional Audio NDK to the Android NDK installation folder. Ensure that all the items in the Professionalaudio_ndk \platforms folder are copied to the C:\android-ndk-r9\platforms folder.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

4.2.2. Creating the Custom Processing Module

1. In your project folder, create a folder named jni.

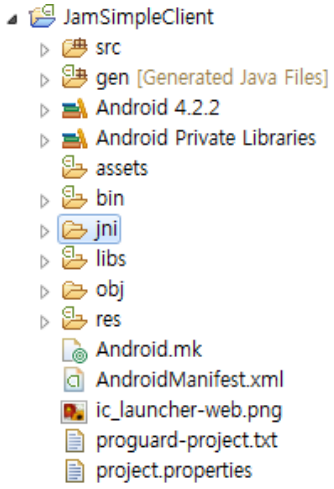

Figure 8: jni folder in Eclipse

2. Implement the following 2 interfaces:

- IAPAInterface: An interface for the frameworks.
- IJackClientInterface: An interface for creating JACK clients.

3. Create a class named APAWave and declare `DECLARE_APA_INTERFACE()` at the end, as shown below:

```
#include <stdio.h>
#include <IAPAInterface.h>
#include "JackSimpleClient.h"
#include "APACommon.h"

namespace android {
	class APAWave : IAPAInterface {
		public:
		APAWave();
		virtual ~APAWave();
		int init();
		int sendCommand(const char* command);
		IJackClientInterface* getJackClientInterface();
		int request(const char* what, const long ext1, const long capacity,
size_t &len, void* data);
	private:
		JackSimpleClient mSimpleClient;
	 };
	DECLARE_APA_INTERFACE(APAWave)
};
```

- When you call `SapaService.register()`, it calls `IAPAInterface.init()` for object creation initialization.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

- When you call SapaProcessor.sendCommand(), it calls IAPAInterface.sendCommand().
- When you call SapaProcessor.queryData(), it calls IAPAInterface.request().
- The what and extra parameters in a query statement are sent as ext1.

4. Create a class that implements the IJackClientInterface interface.

```
#ifndef ANDROID_JACK_SIMPLE_CLIENT_H
#define ANDROID_JACK_SIMPLE_CLIENT_H
#include <jack/jack.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <signal.h>
#include <unistd.h>
#include "IJackClientInterface.h"
#include "APACommon.h"

namespace android {
   class JackSimpleClient: public IJackClientInterface {
#define SIZE_SINE_TABLE (1024)
      public:
            JackSimpleClient();
            virtual ~JackSimpleClient();
            int setUp(int argc, char *argv[]);
            int tearDown();
            int activate();
            int deactivate();
            int transport(TransportType type);
            int sendMidi(char* midi);
      private:
            static int processSine (jack_nframes_t nframes, void *arg);

      private:
            jack_port_t * outPort;
            static jack_client_t *jackClient; // jack client object

            // members for sine tuning
            float sineTable[SIZE_SINE_TABLE];
   };
   };

#endif // ANDROID_JACK_SIMPLE_CLIENT_H
```

- When you call SapaService.register, it calls setup() after IAPAInterface.init(). If you have passed arguments to the constructor while creating SapaProcessor, argv[] of setup() is passed.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

- When you close a processing module by calling `SapaService.unregister()`, it calls `teardown()`.
- When you call `SapaProcessor.activate()`, it calls `activate()`.
- When you call `SapaProcessor.deactivate()`, it calls `deactivate()`.
- When you call `SapaProcessor.setTransportEnabled()`, it calls `transport()`.

5. Create a CUSTOM processing module in a file named libwave.so. Create a makefile as shown below:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := wave
LOCAL_SRC_FILES := \
       wave.cpp \
       JackSimpleClient.cpp

LOCAL_CPP_FEATURES := exceptions
LOCAL_MODULE_TAGS := eng optional
LOCAL_LDLIBS := -ljack -llog
LOCAL_ARM_MODE := arm
LOCAL_CFLAGS := -g
include $(BUILD_SHARED_LIBRARY)
```

6. Implement the APAWave class.

```
#include <string.h>
#include "wave.h"
#include <stdio.h>
#include "mylog.h"

namespace android {

       IMPLEMENT_APA_INTERFACE(APAWave)

       APAWave::APAWave(){
       }

       APAWave::~APAWave(){
       }

       int APAWave::init(){
              LOGD("wave.so initialized");
              return APA_RETURN_SUCCESS;
       }

       int APAWave::sendCommand(const char* command){
           LOGD("APAWave send command [%s]\n", command);
           return APA_RETURN_SUCCESS;
       }

       IJackClientInterface* APAWave::getJackClientInterface(){
           return &mSimpleClient;
       }

       int APAWave::request(const char* what, const long ext1, const long capacity,
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

```
size_t &len, void*data)
	{
		return APA_RETURN_SUCCESS;
	}

};
```

7. APAWave includes JackSimpleClient. The `getJackClinetInterface()` method returns APAWave. You can implement your class relationship here.

```
IJackClientInterface* APAWave::getJackClientInterface(){
	return &mSimpleClient;
}
```

8. Create the JACK client with jack_client_open() and implement the `setup()` method for initializing. Register a process function to generate the sine wave with jack_set_process_callback. Because sounds are output for the speaker, ports are connected to the system left speaker through jack_connect.

```
int JackSimpleClient::setUp (int argc, char *argv[])
{
	LOGD("setUp argc %d", argc);
	for(int i = 0;i< argc; i++){
		LOGD("setup argv %s", argv[i]);
	}


	// make a alias
	char* name = strrchr(argv[0], '.');
	if(name == NULL){
		name = argv[0];
	}
	jackClient = jack_client_open (name, JackNullOption, NULL, NULL);
	if (jackClient == NULL) {
		return APA_RETURN_ERROR;
	}

	jack_set_process_callback (jackClient, processSine, this);

	outPort = jack_port_register (jackClient, "out",
				JACK_DEFAULT_AUDIO_TYPE,
				JackPortIsOutput, 0);
	if(outPort == NULL){
		return APA_RETURN_ERROR;
	}

	return APA_RETURN_SUCCESS;
}
```

9. The JACK client is in standby mode. Activate it by calling the `activate()` method.

```
int JackSimpleClient::activate(){
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

```
    jack_activate (jackClient);
    LOGD("JackSimpleClient::activate");

    const char **systemInputs = jack_get_ports (jackClient, NULL, NULL,
                            JackPortIsPhysical|JackPortIsInput);
    if (systemInputs == NULL) {
        LOGD("system input port is null\n");
        return APA_RETURN_ERROR;
    }

    jack_connect (jackClient, jack_port_name (outPort), systemInputs[0]);

    free (systemInputs);

    return APA_RETURN_SUCCESS;
}
```

The above sample activates the Jack client and reconnects to the port. When you call `activate()`, the process function is called after a certain time interval.

10. Implement the processSine function to create a sine wave sound.

```
int JackSimpleClient::processSine (jack_nframes_t frames, void *arg)
{
    JackSimpleClient *thiz = (JackSimpleClient*)arg;
    jack_default_audio_sample_t *out1 =
          (jack_default_audio_sample_t*)jack_port_get_buffer (thiz->outPort, frames);

    for(unsigned int i=0; i<frames; i++ )
    {
        out1[i] = thiz->sineTable[i%SIZE_SINE_TABLE];
    }

    return 0;
}
```

11. Implement the `teardown()` method.

```
int JackSimpleClient::tearDown(){
    jack_client_close (jackClient);
    return APA_RETURN_SUCCESS;
}
```

12. To build the module, type the following command in the command line:

    #>ndk-build –B (which is incorporated into Android NDK)

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

When you debug, end the process with the processing module and start the debugger. If you debug on Eclipse without ending this process, the UI process ends, but the processing module does not.

4.3. Distributing Type 3 Applications

This section uses the same sine wave example as elsewhere in this document. Replace the appropriate values with your own.

To make your custom processing modules available to other applications:

1. Create an Android project.

- Project Name: SapaSimplePlugin
- Package Name: com.samsung.android.sdk.professionalaudio.sample.simpleplugin
- Activity: No need to create

2. Add the SapaSimplePlugin jni folder.

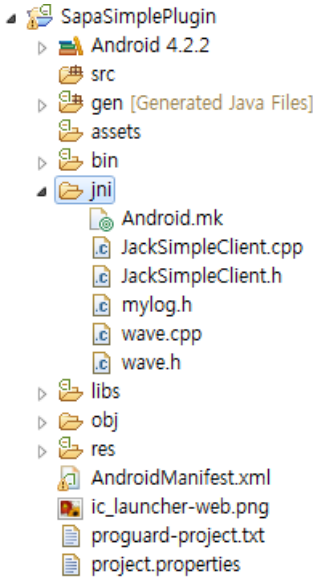

Figure 9: jni folder in Eclipse

3. Edit the Android make file (Android.mk) located in the jni folder.

```
LOCAL_MODULE    := simple_sine
```

4. Create an SO file using the ndk-build command.

The libsimple_sine.so file is created, which indicates a successful compile.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

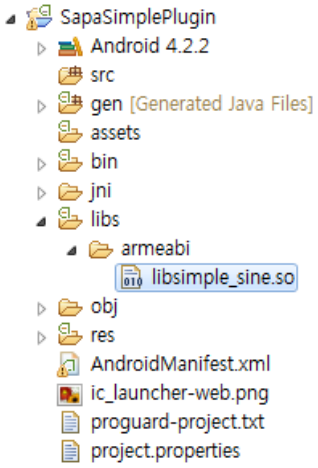

Figure 10: libs folder in Eclipse

5. In the Android manifest file, add a service tag and a meta tag under the application tag.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.samsung.android.sdk.professionalaudio.sample.simpleplugin"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
      <service

android:name="com.samsung.android.sdk.professionalaudio.sample.simpleplugin"
            android:label="SapaSimplePlugin" >
            <intent-filter>
                <action
android:name="com.samsung.android.sdk.professionalaudio.plugin.PICK" />
            </intent-filter>
        </service>
        <meta-data       android:name="SapaSimplePlugin"       android:value="1;Simple
v1.0;SapaSimplePlugin;libsimple_sine.so;;a simple plugin generating a sine wave" />

    </application>

</manifest>
```

6. Include the following intent to enable it to be searched as a plug-in.

```
<intent-filter>
    <action android:name="com.samsung.android.sdk.professionalaudio.plugin.PICK" />
```

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

```
</intent-filter>
```

7. In the android:label of the service tag, list your plug-ins. Use a semicolon as a delimiter to add multiple plug-ins.

```
android:label=" SapaSimplePlugin"
```

8. Add your plug-in meta-data.

```
<meta-data android:name="SapaSimplePlugin" android:value=""/>
```

The name attribute of the meta-data tag should be identical to the android:label attribute of the service tag. Use a semicolon as a delimiter in the following format.

[version];[version name];[plug-in name];[SO file name];[setup argument];[description of the plug-in]

Versions have numeric values and the rest are strings.

9. Compile and generate the APK.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

4.4. Issues

When an activity was in the background (After calling the onPause()), It can be stopped suddenly and forcely by the Android System. Then, The processing module what you registered will working well. Because it is working in a new process. But, If you want to stop it, You need to do something in the onPuase(). It can be unregistering the processing module, and releaseing the related resources.

Copyright © Samsung Electronics Co., Ltd. All rights reserved.

Copyright

Copyright © 2013 Samsung Electronics Co. Ltd. All Rights Reserved.

Though every care has been taken to ensure the accuracy of this document, Samsung Electronics Co., Ltd. cannot accept responsibility for any errors or omissions or for any loss occurred to any person, whether legal or natural, from acting, or refraining from action, as a result of the information contained herein. Information in this document is subject to change at any time without obligation to notify any person of such changes.

Samsung Electronics Co. Ltd. may have patents or patent pending applications, trademarks copyrights or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give the recipient or reader any license to these patents, trademarks copyrights or other intellectual property rights.

No part of this document may be communicated, distributed, reproduced or transmitted in any form or by any means, electronic or mechanical or otherwise, for any purpose, without the prior written permission of Samsung Electronics Co. Ltd.

The document is subject to revision without further notice.

All brand names and product names mentioned in this document are trademarks or registered trademarks of their respective owners.

For more information, please visit http://developer.samsung.com/