대외비

Professional Audio

Programming Guide

Version 2.1.5

Table of Contents

대외비

1. Overview

Professional Audio allows you to create virtual instrument applications with Android. You can connect 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. There is no need to 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 Musical Instrument Digital Interface (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.

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.
- Symmetric multiprocessing (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.

JACK is 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.

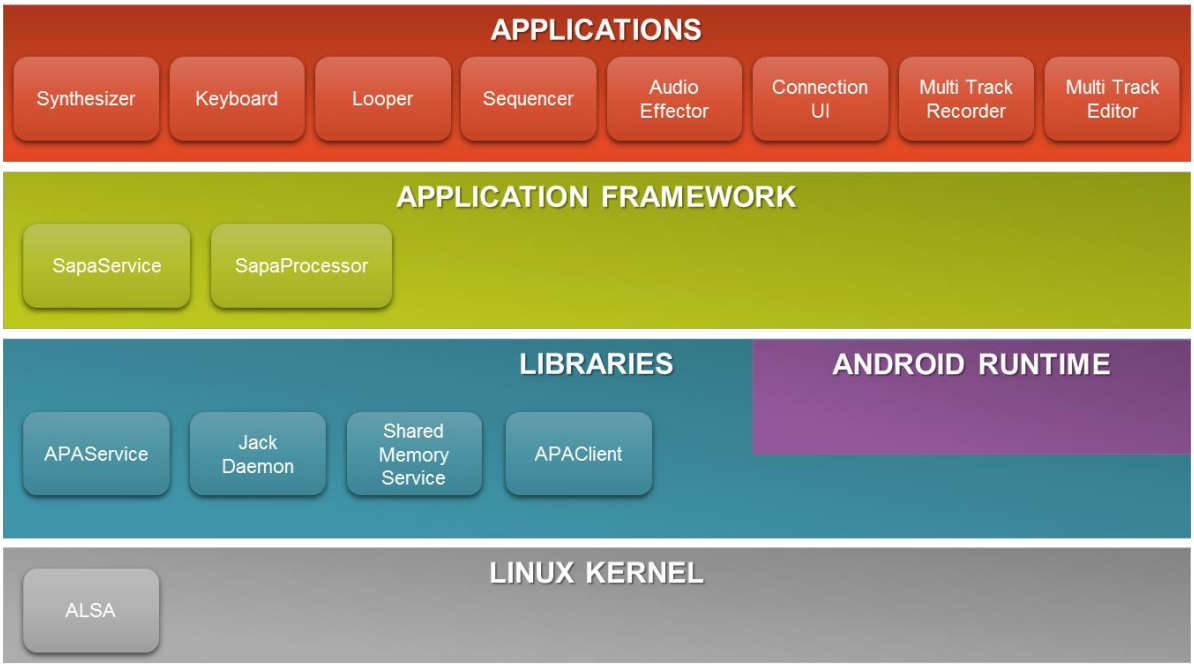

Figure 1: Professional Audio architecture

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.
- **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.
- **Type3:** Include processing modules only. You can develop and distribute your own sound processing modules without UIs in APKs.

※ Type2 and Type3 are deprecated after 2.1.4

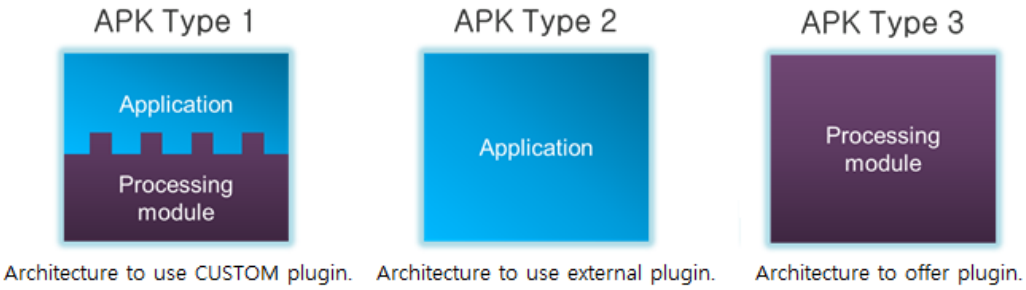

Figure 2: Application types

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.

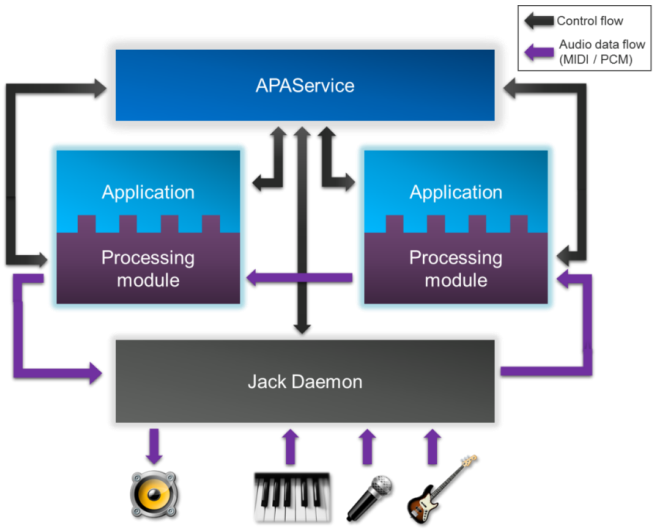

Figure 3: Data flow between 2 type1 applications

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.

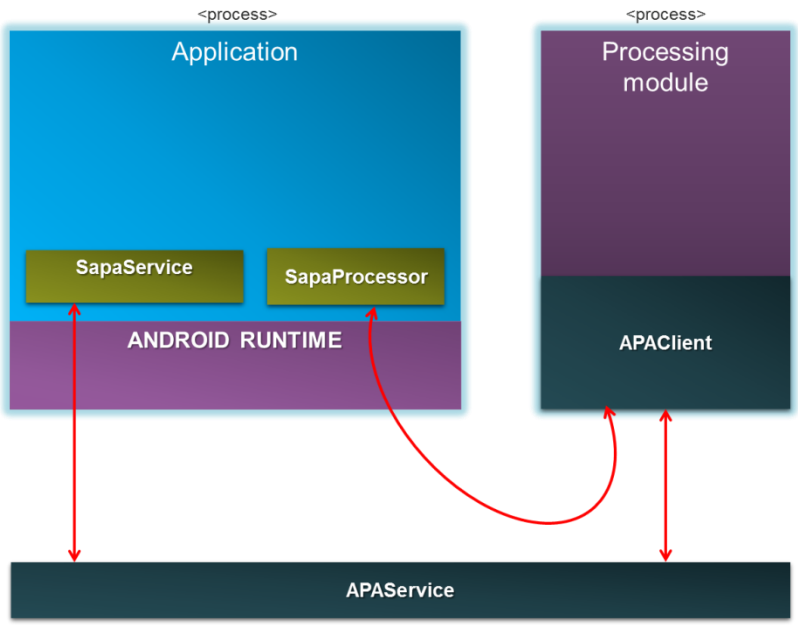

Figure 4: Processes for application and processing module

From the API_LEVEL 22 and over, The processing module is loaded within the Application Process not a separate process. So, The processing module can link and use a shared library. At this case, You should set the minSdkVersion as 22 and over.

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
in: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.

Input Port	Description
	• If an earphone is connected, this port is connected to the earphone microphone. • If the USB device is connected, this port could be connected with the USB device. System:capture_1 is the physical port of this port, but using it is not recommended. Please use in:capture_1 instead.
in: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 capture_1. • If an earphone is connected, this port gets the same input as capture_1. • If the USB device is connected, this port could be connected with the USB device. System:capture_2 is the physical port of this port, but using it is not recommended. Please use in:capture_2 instead.
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
out: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. • If the USB device is connected, this port could be connected with the USB device. System:playback_1 is the physical port of this port, but using it is not recommended. Please use out:playback_1 instead.
out: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. • If the USB device is connected, this port could be connected with the USB device. System:playback_2 is the physical port of this port, but using it is not recommended. Please use out:playback_2 instead.
loopback:playback_1	A virtual port that cannot be connected to any devices. This port is connected to

Output Port	Description
	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 own plug-ins to be used 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.getAllPlugin()` 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.

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 Type 3 Applications.

1.3. Class Diagram

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

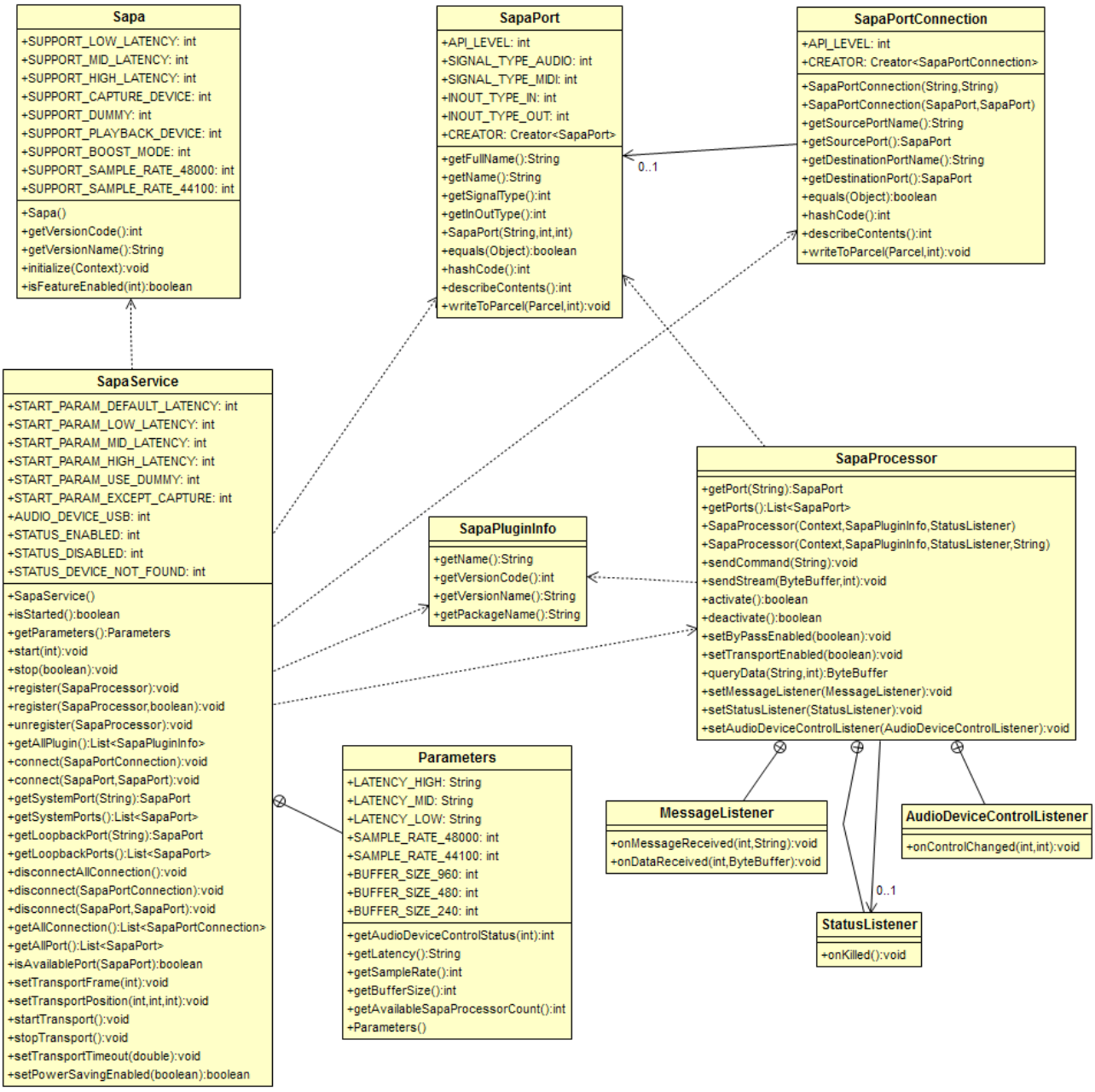

Figure 5: Professional Audio classes and interfaces

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.
- **SapaPluginInfo:** Contains plug-in information.
- **SapaPort:** Provides port functionality.

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

1.7. Installing the Package for Eclipse

To install Professional Audio in Eclipse, add the following files and folder to the libs folder:

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

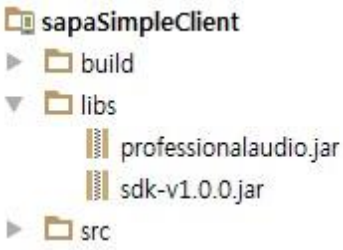

Figure 6: libs folder in Eclipse

The following permission has to be specified in the AndroidManifest.xml file to initialize Professional Audio.

```
<uses-permission android:name=
"com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY"/>
```

If you don’t add the permission,

- o Android 4.4.2 (KitKat) and above: SecurityException is thrown and your application won’t work.
- o Prior to Android 4.4.2 (KitKat): No exception and the application will work properly.

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

2. sapaSimpleClient

Hello Sapa is a sample program that uses a simple client to generate a sine wave sound.

- Project name: SapaPluginClient
- Package name: com.samsung.android.sdk.professionalaudio.sample.simpleclient
- Activity name: SapaSimpleClientActivity

Hello Sapa implements the following features:

1. Initializes the Professional Audio package and make SapaProcessor.

```
    Sapa sapa = new Sapa();
    sapa.initialize(this);
    mService = new SapaService();
    mService.stop(true);
    mService.start(SapaService.START_PARAM_DEFAULT_LATENCY);
    mClient = new SapaProcessor(this, null, new SapaProcessor.StatusListener() {

        @Override
        public void onKilled() {
            Log.v(TAG, "SapaSimpleClient will be closed. because of the SapaProcessor was closed.");
            mService.stop(true);
            finish();
        }
    });
    mService.register(mClient, false);

} catch (SsdkUnsupportedException e) {
    e.printStackTrace();
    Toast.makeText(this, "not support professional audio",
            Toast.LENGTH_LONG).show();
    finish();
    return false;
} catch (InstantiationException e) {
    e.printStackTrace();
    Toast.makeText(this, "fail to instantiate", Toast.LENGTH_LONG)
            .show();
    finish();
    return false;
} catch (AndroidRuntimeException e) {
    e.printStackTrace();
    Toast.makeText(this, "fail to start", Toast.LENGTH_LONG)
            .show();
    finish();
    return false;
}
```

2. Activates the SapaProcessor instance.

```
mPlayButton = (Button) this.findViewById(R.id.play_button);
mPlayButton.setOnClickListener(new Button.OnClickListener() {
    public void onClick(View v) {
        if (mClient != null) {
            mClient.activate();
            mPlayButton.setText(R.string.playing_sound);
            mPlayButton.setEnabled(false);
        }
    }
});
```

The complete code of the sample application is given below.

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

import android.app.Activity;
import android.os.Bundle;
import android.util.AndroidRuntimeException;
import android.util.Log;
import android.view.View;
import android.widget.Button;
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 SapaSimpleClientActivity extends Activity {

       private final static String TAG = "SapaSimpleClient";

       Button mPlayButton, mActivateButton, mDeactivateButton;

       SapaService mService;
       SapaProcessor mClient;

       @Override
       public void onCreate(Bundle savedInstanceState) {
              super.onCreate(savedInstanceState);
              setContentView(R.layout.activity_apasimple_client);

              try {
                     Sapa sapa = new Sapa();
                     sapa.initialize(this);
                     mService = new SapaService();
                     mService.stop(true);
                     mService.start(SapaService.START_PARAM_DEFAULT_LATENCY);
                     mClient = new SapaProcessor(this, null, new
SapaProcessor.StatusListener() {

                           @Override
                           public void onKilled() {
                                  Log.v(TAG, "SapaSimpleClient will be closed. because
of the SapaProcessor was closed.");
```

```
                                mService.stop(true);
                                finish();
                        }
                });
                mService.register(mClient);

        } catch (SsdkUnsupportedException e) {
                e.printStackTrace();
                Toast.makeText(this, "not support professional audio",
                                Toast.LENGTH_LONG).show();
                finish();
                return;
        } catch (InstantiationException e) {
                e.printStackTrace();
                Toast.makeText(this, "fail to instantiate", Toast.LENGTH_LONG)
                                .show();
                finish();
                return;
        } catch (AndroidRuntimeException e){
                e.printStackTrace();
                Toast.makeText(this, "fail to start", Toast.LENGTH_LONG)
                                .show();
                finish();
                return;
        }

        mPlayButton = (Button) this.findViewById(R.id.play_button);
        mPlayButton.setOnClickListener(new Button.OnClickListener() {
                public void onClick(View v) {
                        if (mClient != null) {
                                mClient.activate();
                                mPlayButton.setText(R.string.playing_sound);
                                mPlayButton.setEnabled(false);
                        }
                }
        });

        mActivateButton = (Button) this.findViewById(R.id.button_activate);
        mActivateButton.setOnClickListener(new Button.OnClickListener() {
                public void onClick(View v) {
                        if (mClient != null)
                                mClient.activate();
                }
        });

        mDeactivateButton = (Button) this.findViewById(R.id.button_deactivate);
        mDeactivateButton.setOnClickListener(new Button.OnClickListener() {
                public void onClick(View v) {
                        if (mClient != null)
                                mClient.deactivate();
                }
        });
}

@Override
protected void onDestroy() {
        super.onDestroy();
        if(mService != null){
                if(mClient != null){
```

```
                        mService.unregister(mClient);
                    }
                    mService.stop(true);
                }
            }
        }
```

Use the permission in Manifest.xml.

```
<uses-permission
android:name="com.samsung.android.sdk.professionalaudio.permission.START_MONITOR_SERVICE"/>
```

The monitor service checks whether the Jack daemon is running or not.

You can stop the Jack daemon through it to save the battery power.

```
<uses-permission
android:name="com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY"
/>
```

The permission is defined in the Samsung SDK. It is used for statistics of Professional Audio SDK usages. It is mandatory for the SDK. It is not for the user.

```
<uses-permission
android:name="com.samsung.android.sdk.professionalaudio.permission.USE_CONNECTION_SERVICE" />
```

When you use the AudioConnectionService, you have to include this permission. It is for granting access to the audio connection service.

For more information, see SapaPluginSimpleClient.zip.

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 the `initialize()` method to initialize 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
    sapa.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

- 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);
```

4. Using the Professional Audio Package

4.1. Creating a Type 1 Application using the NDK

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

The processing module is loaded on the separated process not UI process. So, you can use only a static library not shared.

You can use the SapaProcessor class in the SDK to communicate with a UI(Java).

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

4.1.1. Setting up the Environment

Download and install the 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 environment variables on your computer; In the window for setting environment variables, in the **Path** textbox, add C:\android-ndk-r9.
4. Test your settings in the command line as shown below:

```
c:₩>ndk-build --version
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

This program built for i586-pc-mingw32
```

Figure 7: Checking if the installation is successful

5. Copy and paste the Professional Audio NDK to the Android project folder. Ensure that all the items in the Professionalaudio_ndk\platforms\android-<version>\arch-<arch>\usr\ folder are copied to the <project root>\jni\apa\.

4.1.2. Creating the Custom Processing Module

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

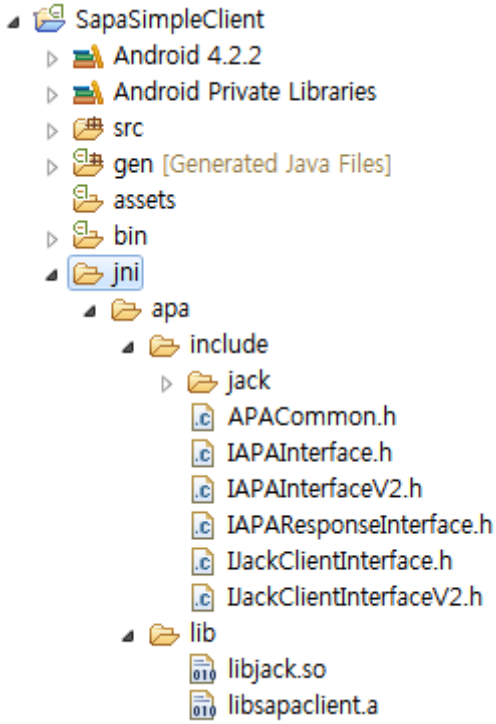

Figure 8: jni folder in Eclipse

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.
- When you call `SapaProcessor.sendCommand()`, it calls `IAPAInterface.sendCommand()`.
- When you call `SapaProcessor.queryData()`, it calls `IAPAInterface.request()`.
- The 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.
- 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    := libsapaclient
LOCAL_SRC_FILES := apa/lib/libsapaclient.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := libjack
LOCAL_SRC_FILES := apa/lib/libjack.so
include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MULTILIB := 32
# The module name have to be “wave”.
# It could be changable when the module is provided as a plug-in.
LOCAL_MODULE    := wave
```

```
LOCAL_C_INCLUDES := $(LOCAL_PATH)/apa/include
LOCAL_SRC_FILES := \
    wave.cpp \
    JackSimpleClient.cpp
LOCAL_CPP_FEATURES := exceptions
LOCAL_MODULE_TAGS := eng optional
LOCAL_LDLIBS := -llog
LOCAL_SHARED_LIBRARIES := libjack
LOCAL_STATIC_LIBRARIES := libsapaclient
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,
size_t &len, void*data)
	{
		return APA_RETURN_SUCCESS;
	}

};
```

7. APAWave includes JackSimpleClient. The `getJackClientInterface()` method returns APAWave. You can implement your class relationships 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]);
        }

        // You have to use argv[0] as the jack client name like below.
        jackClient = jack_client_open (argv[0], JackNullOption, NULL, NULL);
        if (jackClient == NULL) {
                return APA_RETURN_ERROR;
        }

        bufferSize = jack_get_buffer_size(jackClient);

        // init the sine table
        for(int i=0; i<bufferSize; i++ )
        {
                sineTable[i] = (float) sin( ((double)i*8/(double)bufferSize) * 2. *
3.14159265 );
        }

        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(){

                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%bufferSize];
        }

            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)

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.

5. Using the Professional Audio App Package

5.1. Registration

When you want to inform the system that your application is an audio application you must modify your AndroidManifest.xml file.

Your application must use two additional permissions:

```
<uses-permission android:name="com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY"/>
<uses-permission android:name="com.samsung.android.sdk.professionalaudio.permission.USE_CONNECTION_SERVICE"/>
```

Your activity must have the following intent-filter.

```
<intent-filter>
    <action android:name="com.samsung.android.sdk.professionalaudio.action.MAIN" />
    <category android:name="com.samsung.android.sdk.professionalaudio.category.LAUNCHER" />
</intent-filter>
```

This indicates that the application is an audio application, but this declaration is not enough. You also need to specify your application using meta-data.

Below you can find attributes on which specification are obligatory and those that are optional accompanied with name of the meta-data object specifying them.

Obligatory	
Application name.	com.samsung.android.sdk.professionalaudio.sapaappinfo.appname
Action starting background service of application.	com.samsung.android.sdk.professionalaudio.sapaappinfo.backgroundservice
Category*	com.samsung.android.sdk.professionalaudio.sapaappinfo.category
Optional	
Name of vendor	com.samsung.android.sdk.professionalaudio.sapaappinfo.vendorname
Name of group of products	com.samsung.android.sdk.professionalaudio.sapaappinfo.productgroupname
Name of application version	com.samsung.android.sdk.professionalaudio.sapaappinfo.version
Information whether application handles more than one instance**	com.samsung.android.sdk.professionalaudio.sapaappinfo.multiinstanceenabled
Information whether application has at least one midi input port	com.samsung.android.sdk.professionalaudio.sapaappinfo.midiinportcount
Information whether	com.samsung.android.sdk.professionalaudio.sapaappinfo.midioutportcount

application has at least one midi output port	
Information whether application has at least one audio input port	com.samsung.android.sdk.professionalaudio.sapaappinfo.audioinportcount
Information whether application has at least one audio output port	com.samsung.android.sdk.professionalaudio.sapaappinfo.audiooutportcount
Description of the application	com.samsung.android.sdk.professionalaudio.sapaappinfo.description

**Possible values: instrument, utility, effect.*

***Possible values: integers*

Example of such file:

```
<activity
    android:name="com.samsung.audio.test.testapplication_a.MainActivity"
    android:label="@string/app_name" >

    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.samsung.android.sdk.professionalaudio.action.MAIN" />

        <category android:name="com.samsung.android.sdk.professionalaudio.category.LAUNCHER" />
    </intent-filter>

    <meta-data
        android:name=" com.samsung.android.sdk.professionalaudio.sapaappinfo.category"
        android:value="utility" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.version"
        android:value="v1.0" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.vendorname"
        android:value="samsung" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.productgroupname"
        android:value="samples" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.appname"
        android:value="mixer" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.multiinstanceenabled"
        android:value="false" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.backgroundservice"
        android:value="com.samsung.audio.test.testapplication_a.MainService" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.midiinportcount"
        android:value="1" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.midioutportcount"
        android:value="1" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.audioinportcount"
        android:value="0" />
    <meta-data
```

```
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.audiooutportcount"
        android:value="0" />
    <meta-data
        android:name="com.samsung.android.sdk.professionalaudio.sapaappinfo.description"
        android:value="This is a sample application" />

</activity><service
    android:name="com.samsung.audio.test.testapplication_a.MainService"
    android:enabled="true"
    android:exported="true"
    android:permission="com.samsung.android.sdk.professionalaudio.permission.USE_CONNECTION_SERVICE" >

    <intent-filter>
        <action android:name="com.samsung.audio.test.testapplication_a.MainService" />
    </intent-filter>

</service>
```

5.2. Background Service

5.2.1. Why is it needed?

Almost every audio application works while they are not visible, this work shall be maintained in a service. If not, activity may be stopped when it is not in foreground.It is also to maintain ports registered if service is needed.

5.2.2. Handling connection to Audio Connection Service

Professional Audio provides a class, SapaAppService to handle connection to Audio Connection Service. In this part you can see how to use it in different types of applications.

5.2.2.1 Single-instance-application without launcher

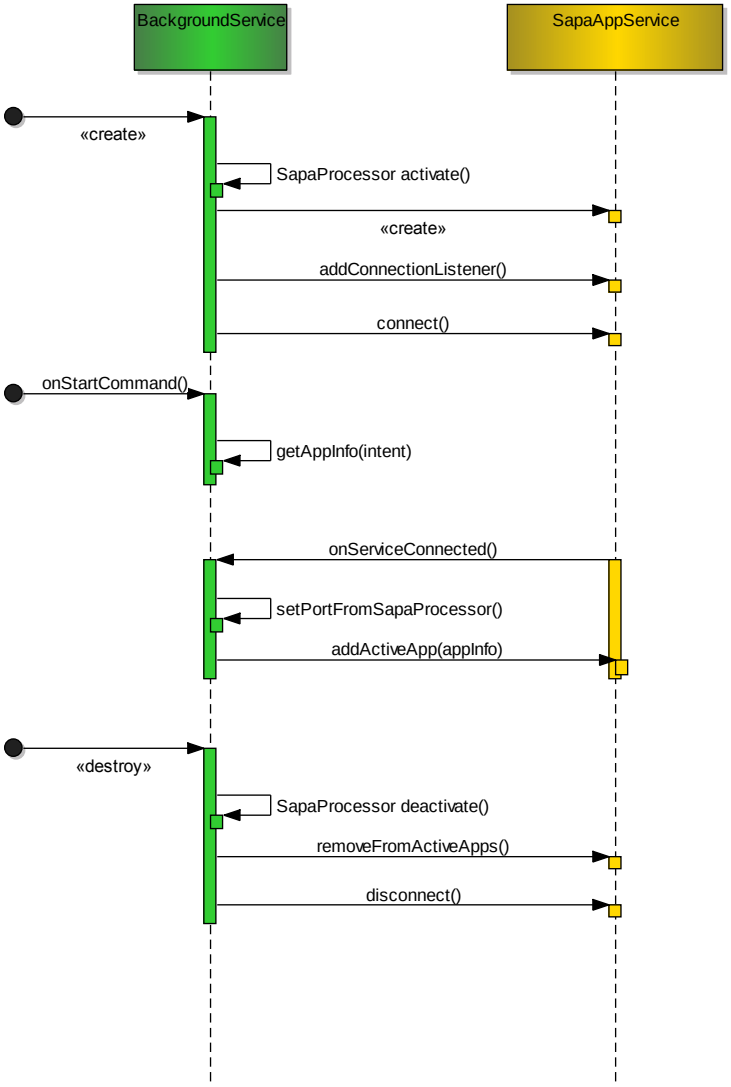

Figure 11: Single-instance-application without launcher

Example code of such a service :

```
@Override
public void onCreate() {
    Log.d(TAG, "onCreate");
    super.onCreate();
    this.mIsSapaAppServiceConnected = false;
    this.mSapaAppService = new SapaAppService(this);
    try {
        this.mSapaAppService.setConnectionListener(this);
    } catch (TooManyListenersException e) {
        Log.e(TAG, "Listener was already set");
    }
    this.mSapaAppService.connect();
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    SapaAppInfo info = SapaAppInfo.getAppInfo(intent);
```

```
    if (info != null) {
        this.mMyInfo = info;
    }
    return super.onStartCommand(intent, flags, startId);
}

@Override
public void onDestroy() {
    Log.d(TAG, "onDestroy");
    try {
        this.mSapaAppService.removeFromActiveApps(this.mMyInfo.getApp());
    } catch (SapaConnectionNotSetException e) {
        Log.e(TAG,"Instance could not be removed from active list because of connection exception.");
    }
    this.mSapaAppService.disconnect();
    super.onDestroy();
}

@Override
public void onServiceConnected() {
    Log.d(TAG, "onServiceConnected");
    this.mIsSapaAppServiceConnected = true;
    try {
        if (this.mMyInfo == null) {
            this.mMyInfo = this.mSapaAppService.getInstalledApp(getPackageName());
        }
        if (this.mMyInfo != null) {
            //SapaProcessor needs to be registered and activated before calling this method
            this.mMyInfo.setPortFromSapaProcessor(this.mSapaProcessor);
            this.mSapaAppService.addActiveApp(this.mMyInfo);
        }
    } catch (SapaConnectionNotSetException e) {
        Log.e(TAG,"App could not be added to active as connection has not been made.");
    }
}

@Override
public void onServiceDisconnected() {

}
```

5.2.2.2 Single-instance-application that is able to work in stand-alone mode

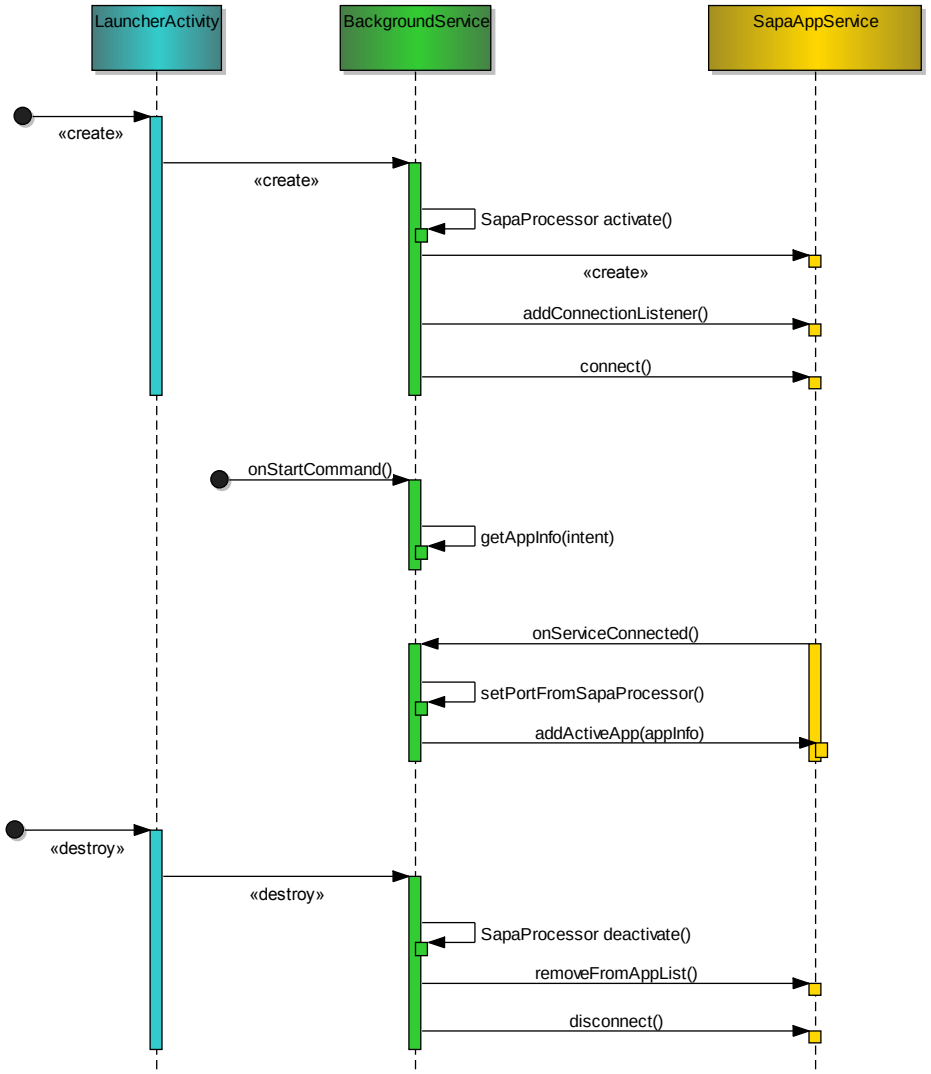

Figure 12: Single-instance-application that is able to work in stand-alone mode

5.2.2.3 Multi-instance-application without launcher

Multi-instance-application without launcher is similar to single-instance-application without launcher. The difference is in the reaction for onStartCommand which is used to activate and deactivate instances. Below you can find a sample code.

Example code of such a service :

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    int state = SapaAppInfo.getState(intent);
```

```
    Log.d(TAG, "onStartCommand state: " + state);
    if (state == SapaAppInfo.STATE_ACTIVATING) {
        SapaAppInfo info = SapaAppInfo.getAppInfo(intent);
        if (info != null) {
            if (this.mIsSapaAppServiceConnected) {
                this.activateInstance(info);
            }else {
                this.mInstancesToBeStarted.add(info);
            }
        }
    } else if (state == SapaAppInfo.STATE_DEACTIVATING) {
        SapaApp sapaApp = SapaAppInfo.getApp(intent);
        deactivateInstance(sapaApp);
    }
    return super.onStartCommand(intent, flags, startId);
}

private void activateInstance(SapaAppInfo info){
    try {
        this.mSapaAppService.addActiveApp(info);
        this.mMyInstances.put(info.getApp(), info);
    } catch (InvalidParameterException e) {
        Log.e(TAG, "onStartCommand " + SapaAppInfo.STATE_ACTIVATING + " InvalidParameterException " +
info.getApp().getInstanceId());
    } catch (ConnectionNotSetException e) {
        Log.e(TAG, "onStartCommand " + SapaAppInfo.STATE_ACTIVATING + " SapaConnectionNotSetException "+
info.getApp().getInstanceId());
    }
}

private void deactivateInstance(SapaApp sapaApp) {
    Log.d(TAG, "\t\tdeactivateInstance( " + sapaApp.getInstanceId() + " )");
    this.mMyInstances.remove(sapaApp);
    try {
        this.mSapaAppService.removeFromActiveApps(sapaApp);
    } catch (SapaConnectionNotSetException e) {
        Log.e(TAG, "Connection problem while deactivating " + sapaApp.getInstanceId());
    }
}

@Override
public void onServiceConnected() {
    this.mIsSapaAppServiceConnected = true;
    for(SapaAppInfo info : this.mInstancesToBeStarted){
        this.activateInstance(info);
    }
}

@Override
public void onServiceDisconnected() {

}
```

5.3. How to convert application holding only one instance to multiinstance application

- You need to modify your AndroidManifest so that in the meta-data multiinstanceenalbed is set to true.
- In onStartCommand you need to read the process value [SapaAppInfo.getState(Intent)]. It can be set to STATE_ACTIVATING or STATE_DEACTIVATING.

    - When STATE_ACTIVATING is called you need to start a new SapaProcessor and store mapping AppInfo.getInstanceId with reference to that SapaProcessor. As always you need to add new active instance of application using addActiveApp method of SapaAppService.
    - When STATE_DEACTIVATING is called you need to unregister the proper SapaProcessor. The SapaApp instance identifying the instance is in the received Intent. You can read it using the method SapaAppInfo.getApp(Intent);

  ! Please note that when the last instance of your application is being deactivated you will receive stopService, so onDestroy method will be called. As always use removeFromActiveApps method for the instance that is being closed.
- When the activity of your application is started you will receive SapaAppInfo, representing the instance of your application to be shown in the starting intent. Remember to set all activity controls according to the instance that is being called.
- Every time an action is called you get information about the instance on which action is to be performed.

Summing up:

You will show only one instance of our application at a time so you do not need new activities. You will need only one background service that will be modified a bit. Your native part will not change because you will send commands to every instance of it via its SapaProcessor.

5.4. Available classes

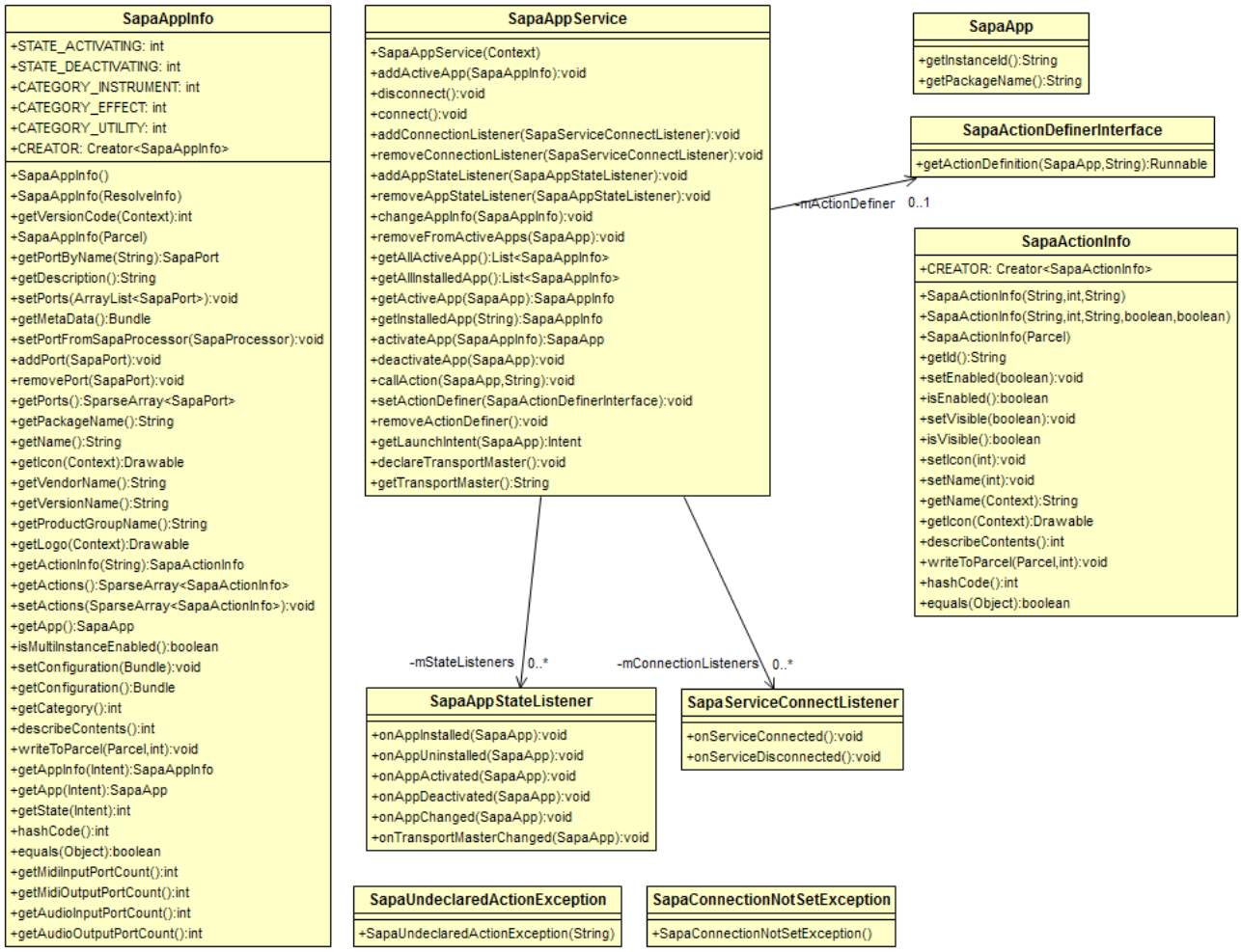

Figure 13: Professional Audio App package classes

5.5. Ports

5.5.1. Declaring ports

When your service is started you can obtain the SapaAppInfo object representing your application in the following way:

```
SapaAppInfo info = SapaAppInfo.getAppInfo(intent);
```

During activation of your application, you shall register your SapaProcessor. After registering the SapaProcessor and before declaring that your application is activated, you shall add registered ports to obtain information. You shall do that in the following way:

```
Sapa sapa = new Sapa();
sapa.initialize(mContext);
```

```
mSapaService = new SapaService();
mSapaProcessor = new SapaProcessor(mContext, null, null);
mSapaService.register(mSapaProcessor);
mSapaProcessor.activate();
info.setPortFromSapaProcessor(mSapaProcessor);
```

*Exception handling was omitted for clarity of the example.

5.6. Listening for changes on Audio Connection Service

Your application may be interested in obtaining information whenever some new information is available on Audio Connection Service. SapaAppStateListener is prepared for it.

This interface enables you to get information whenever:

- Audio application is installed
- Audio application is uninstalled
- Instance of audio application is activated
- Instance of audio application is deactivated
- Information about active instance is changed

Your listener shall be added on AppService using method setAppStateListener(listener: SapaAppStateListener) and removed using removeAppStateListener().

5.7. Handling actions

5.7.1. What is an action?

Your application may have functionalities that you may want to be called from other applications. For example if you have a recorder you may want to expose start record option to other applications so that the button representing start recording in your application can be placed in an activity of another application. Such functionalities shall be declared as actions.

5.7.2. Declaring actions

Actions shall be defined in SapaAppInfo of your active application. Though you can change declared actions while your application is active it is recommended to declare them during activation of application. Actions are declared using the SapaAppInfo class.

5.7.3. Defining actions

Actions are declared in SapaAppInfo but to define them you need to implement SapaActionDefinerInterface and set this definer on SapaAppService. It shall be set before adding action declarations to SapaAppInfo.

5.8. Changing state of activate instance of application

When you want to change some information in the SapaAppInfo of your active application, you need to obtain the SapaAppInfo you want to change, modify the attribute and then using changeAppInfo() method to expose the changes to other applications.

5.9. Saving and loading presets

If your application handles connecting to other applications you may need to save the state of those applications in its preset and be able to restore those applications when loading a preset. This approach is possible because every application shall keep up to date SapaAppInfo instance describing its state on active applications list on the Audio Connection Service.

SapaAppInfo is a Parcelable class. That means that it can be easily saved to for example, a database as an object and then restored.

5.9.1. Saving a preset

1. Save configuration.
2. Save SapaAppInfo objects representing instances of applications that you want to remember (you can obtain that list from service using getAllActiveApp() method from SapaAppService).
3. Save connections.

5.9.2. Loading a preset

1. Set configuration.
2. Activate applications in the saved state using activateApp(SapaAppInfo info) method from SapaAppService.
3. Wait until all applications are activated.
4. Set connections.

Note that it may be impossible to activate all needed applications as some other applications may be opened before. You need to decide whether you want to deactivate all other applications before loading a preset or handle this problem another way.

5.10. Using FloatingController in your application

When you want to use FloatingController in your application you must add it to your layout xml file.

```
<com.samsung.android.sdk.professionalaudio.widgets.FloatingController
    xmlns:control="http://schemas.android.com/apk/res-auto"
    android:id="@+id/jam_control"
    control:type="in_corners"
    control:handle_drawable="@drawable/handle"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:soundEffectsEnabled="false" />
```

The FloatingController consists of two components:

- The scrollable bar with buttons
- The invisible area over which the bar can be dragged and dropped

The image below is a scheme of the widget in "*in_corners*" mode.

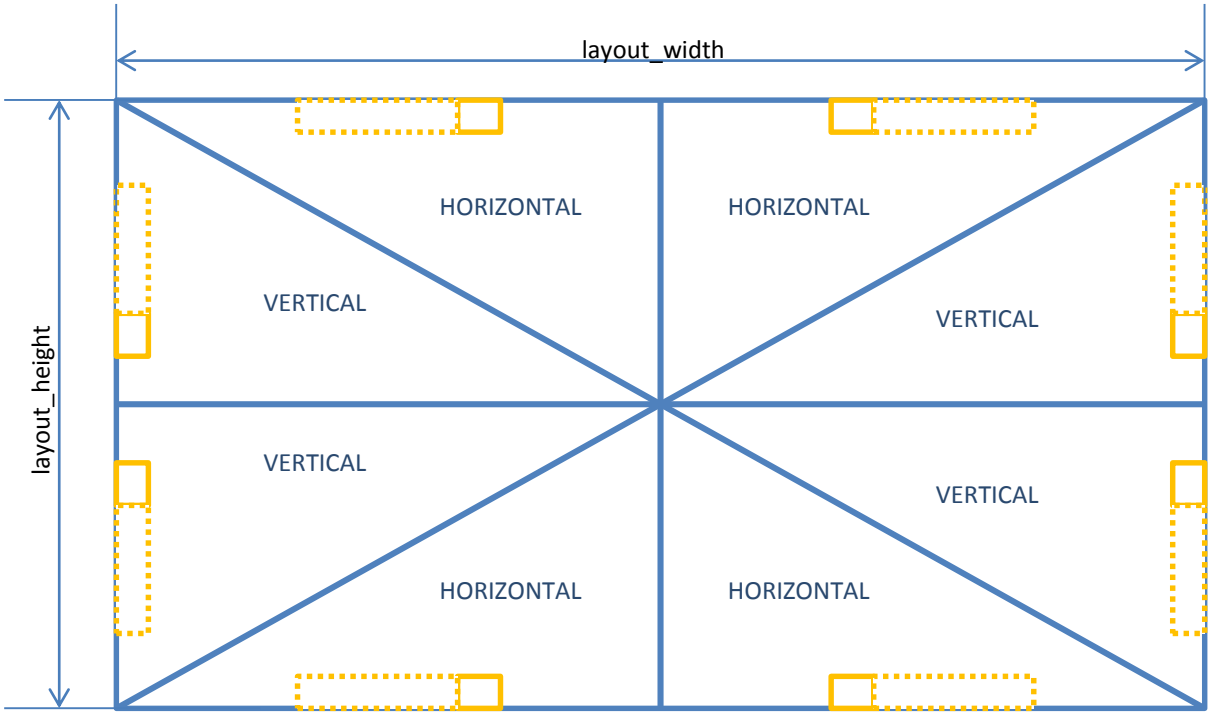

Figure 14: Scheme of the widget in "in_corners" mode

The above image depicts the invisible area of the FloatingController where the bar can be dropped in 8 places. Dropping the bar on some areas make it horizontal and dropping on other areas make it vertical. Arrows show directions of expansion of the bar in different areas.

The *layout_width* and *layout_height* attributes affect basically the invisible area because dimensions of the bar follow the concept of Soundcamp.

It is best to put the FloatingController on the top of the whole layout of application which will prevent the bar from being obscured.

Notice!
In order to use *in_corners* developer should add android:supportsRtl="true" in <application> section in the AndroidManifest.xml

In the example above you can see that width and height are set to match_parent. The size of the FloatingController pertains to the size of the space above in which FloatingController can be moved.

To make the control work, you need to give it access to the SapaAppService of your application. You can do this using the setSapaAppService method on the FloatingController. It is all that is required for basic usage.

5.10.1. Advanced usage

In the above sample, we saw additional attributes, *type* and *handle_drawable*.

The *type* attribute can be either *in_corners* (jam control will be located in corners of screen) or *center_on_edges* (jam_control will be in the middle of edges). The default value for this attribute is *in_corners*.

Initial position of the floating controller can be set with "*bar_alignment*" attribute. For type *in_corners* it can be: topLeft, topRight, bottomLeft or bottomRight. While for type *center_on_edges* it can be: top, right, left or bottom.

Attribute *orientation* works only for type *in_corners* and sets if floating controller should be *vertical* or *horizontal*.

As said before, the bar of FloatingController moves on the invisible area on the top of the hierarchy. Therefore if the developer does not want the FloatingController to obscure other widgets, it is the best to provide the margin of 50 dp to those widgets so that they are drawn next to the bar.

The *handle_drawable* attribute of the FloatingController is expanded if the state "activated=true" which will be called when the widget is expanded.

The developer can specify one single instance application for which an application icon will always be visible in FloatingController when the application is active (only when type="in_corners"). In order to do that, the developer should extend FloatingController and override getMainPackage() method to return the package of the application.

대외비

Copyright

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/