libvoice Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Library Overview

Purpose and Characteristics

The libvoice library provides the APIs for using the Voice Service. The Voice Service has the following features:

- Compression and decompression of voice data (CELP codec)
- Mixing and volume adjustments
- Voice input/output via microphones and headphones

The Voice Service aids in application development by processing microphone input and headphone output for the application. The application uses the inbound port (IPort) and outbound port (OPort) to exchange voice data with the Voice Service. By combining these ports and allotting codec and mixing functions accordingly, the application can create and use a variety of topologies for voice processing.

The libvoice library can be used by applications with relatively simple topologies (such as one using only voice mixing and recording or playback), applications requiring complicated topologies (such as a voice chat over the network by multiple users), as well as various other types of applications.

An application uses libvoice to handle microphone input. Note that a single microphone input cannot currently be used by more than one client. Consequently, an application cannot use the `libAudio(In)` library on its own while libvoice creates `SCEVOICE_PORTTYPE_IN_DEVICE` port.

Technical Specifications

This section describes general libvoice technical specifications.

Ports

libvoice has the following types of ports:

- Audio input device port
- Audio output device port
- Voice input port
- Voice output port
- PCM file-read input port
- PCM file-write output port

The number of connections between ports in one topology cannot exceed 128.

VOICE Ports

Each active running voice input port is bound with one voice decoder in the system, and each active running voice output port is bound with one voice encoder in the system. The voice service currently handles a maximum of 7 active decoders and 1 active encoder at any given time. libvoice internally has a voice encoder sharing mechanism among voice output ports. The number of voice output ports you can create is not limited by the number of maximum active voice encoders in the system.

For voice output ports, the usage of a voice encoder depends on the differences between port characteristics of the voice outputs. The system uses only one encoder if all active voice output ports have the same characteristics. The following example configuration uses one encoder instance:

- Input A, Input B, Input C connected to Output 1 with a bit rate of 3850 and a volume of 1.0.
- Input A, Input B, Input C connected to Output 2 with a bit rate of 3850 and a volume of 1.0.

The system needs more encoders if the voice output ports have different characteristics. The following example configuration uses three encoder instances, one for Output 1 and 2, one for Output 3, and one for Output 4:

- Input A, Input B, Input C connected to Output 1 with a bit rate of 3850 and a volume of 1.0.
- Input A, Input B, Input C connected to Output 2 with a bit rate of 3850 and a volume of 1.0.
- Input A, Input B, Input C connected to Output 3 with a bit rate of 5700 and a volume of 1.0.
- Input A, Input B, Input C connected to Output 4 with a bit rate of 3850 and a volume of 0.5.

When the bit rate of a VOICE type port is changed, all internal buffers are cleared. The application must ensure that the corresponding voice port on the other side of the connection is set to the same type of value. Failure to do so results in distorted audio output.

The supported voice port sampling rate and bit rate are as follows:

- With a bit rate of 3850 bps port, the sampling rate is 8000 Hz.
- With a bit rate of 4560 bps port, the sampling rate is 8000 Hz.
- With a bit rate of 5700 bps port, the sampling rate is 8000 Hz.
- With a bit rate of 7300 bps port the sampling rate is 8000 Hz.

Transmission of Port State

A voice port has five different port states: NULL, idle, ready, buffering, and running. Each port has one associated state that indicates its current status. The following four figures describe in detail how port state is transmitted for a device in port, a device out port, a voice in port, and a voice out port.

Figure 1   Device In Port State

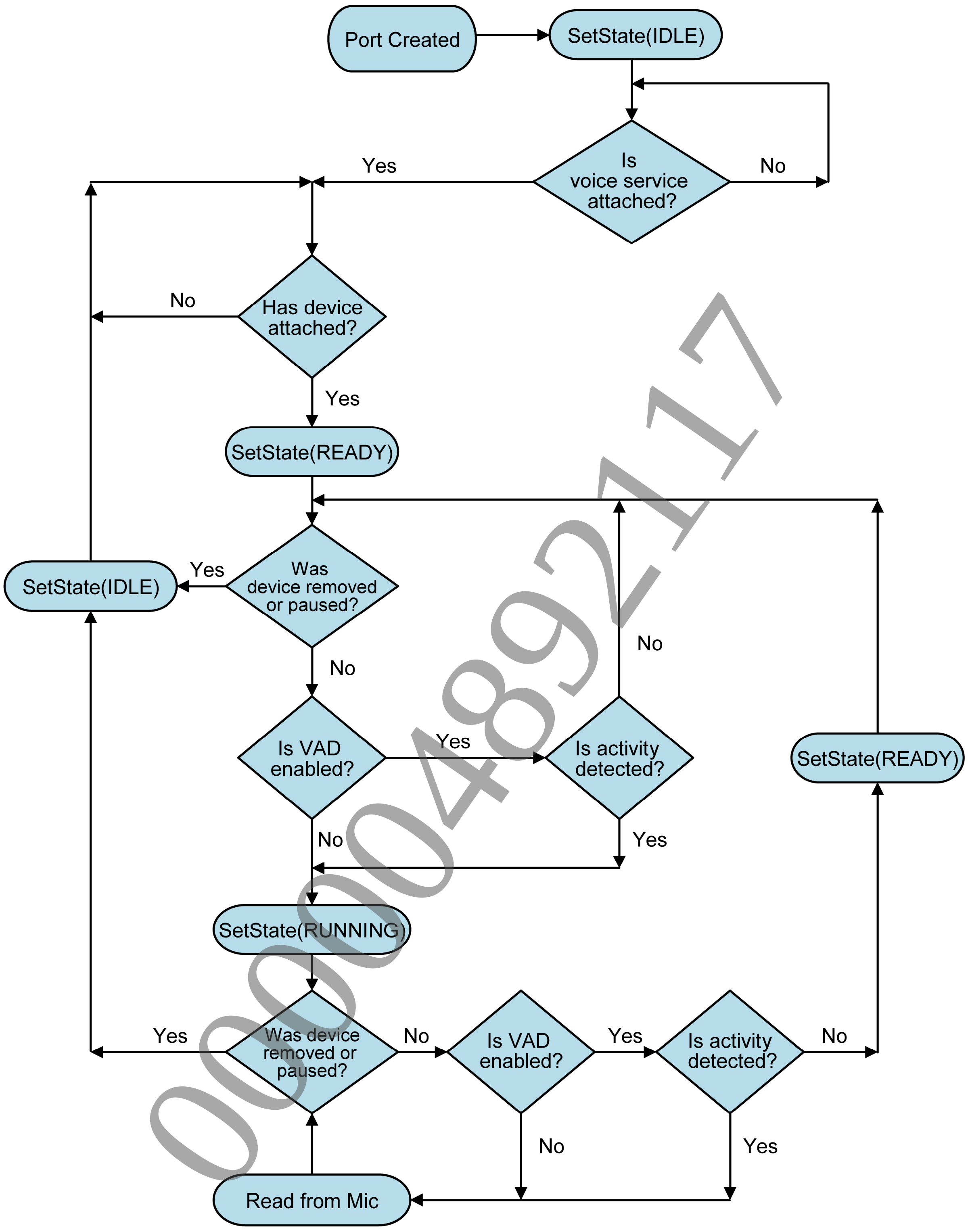


Figure 2　Device Out Port State

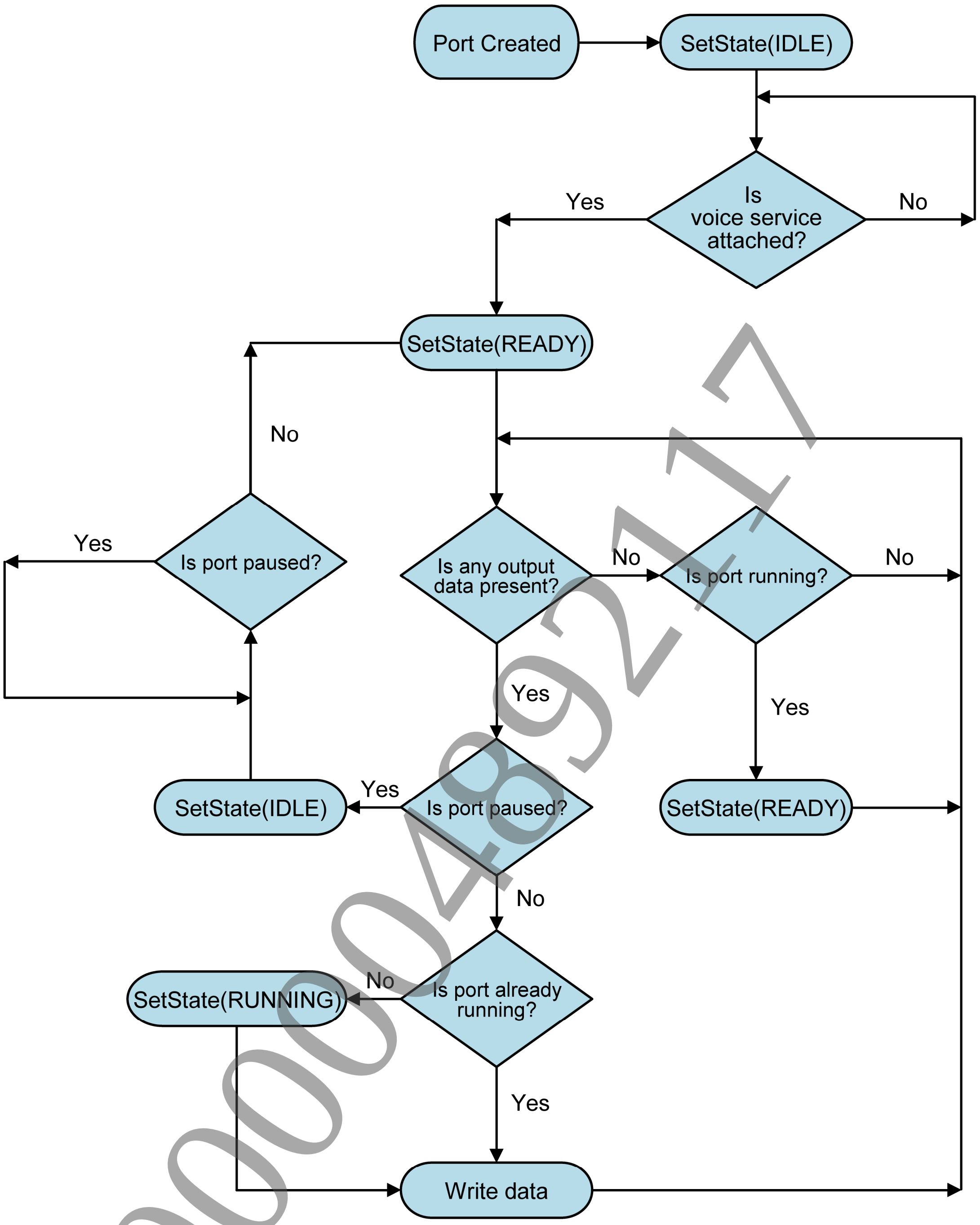


Figure 3 Voice Input Port State

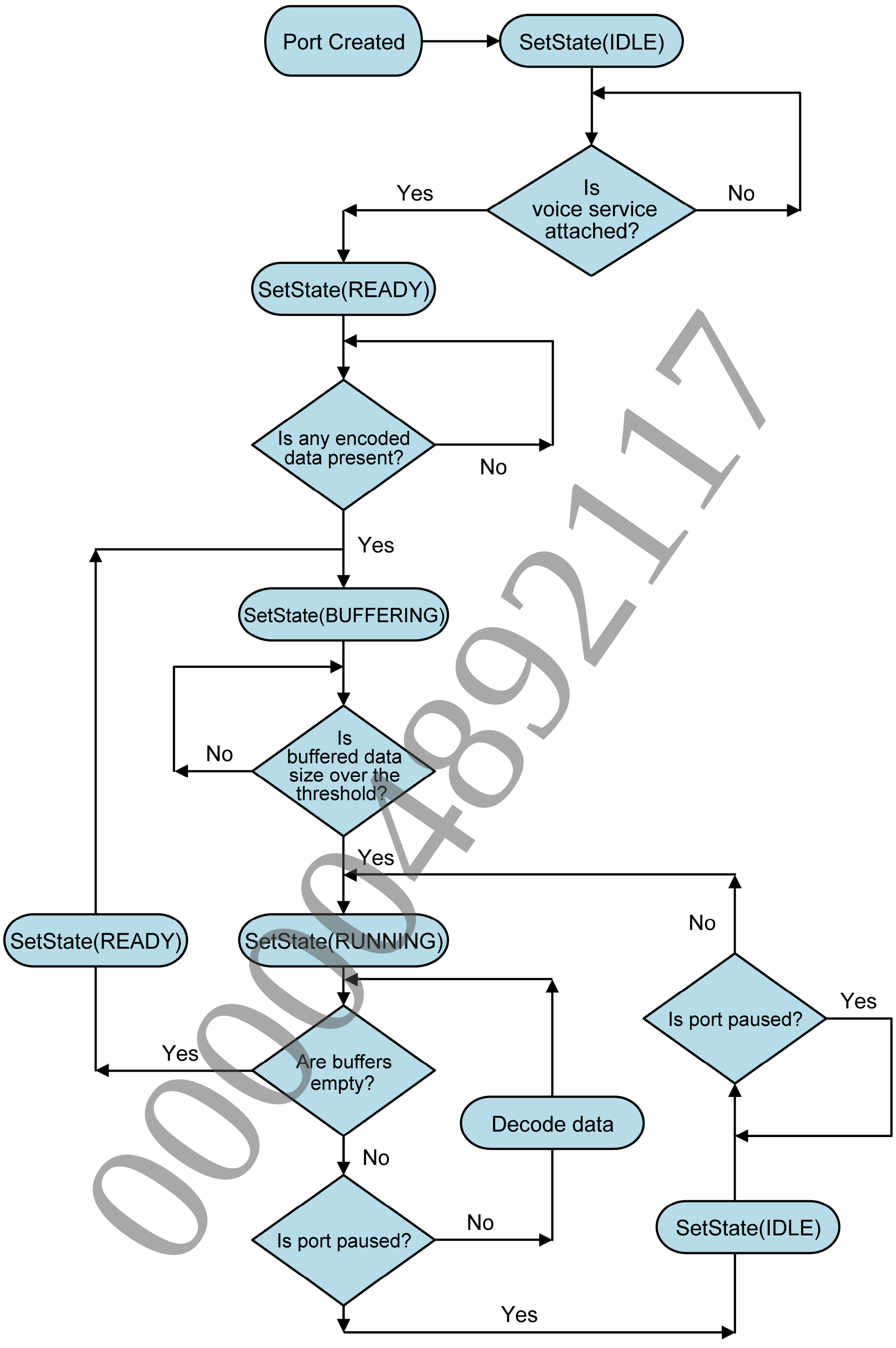
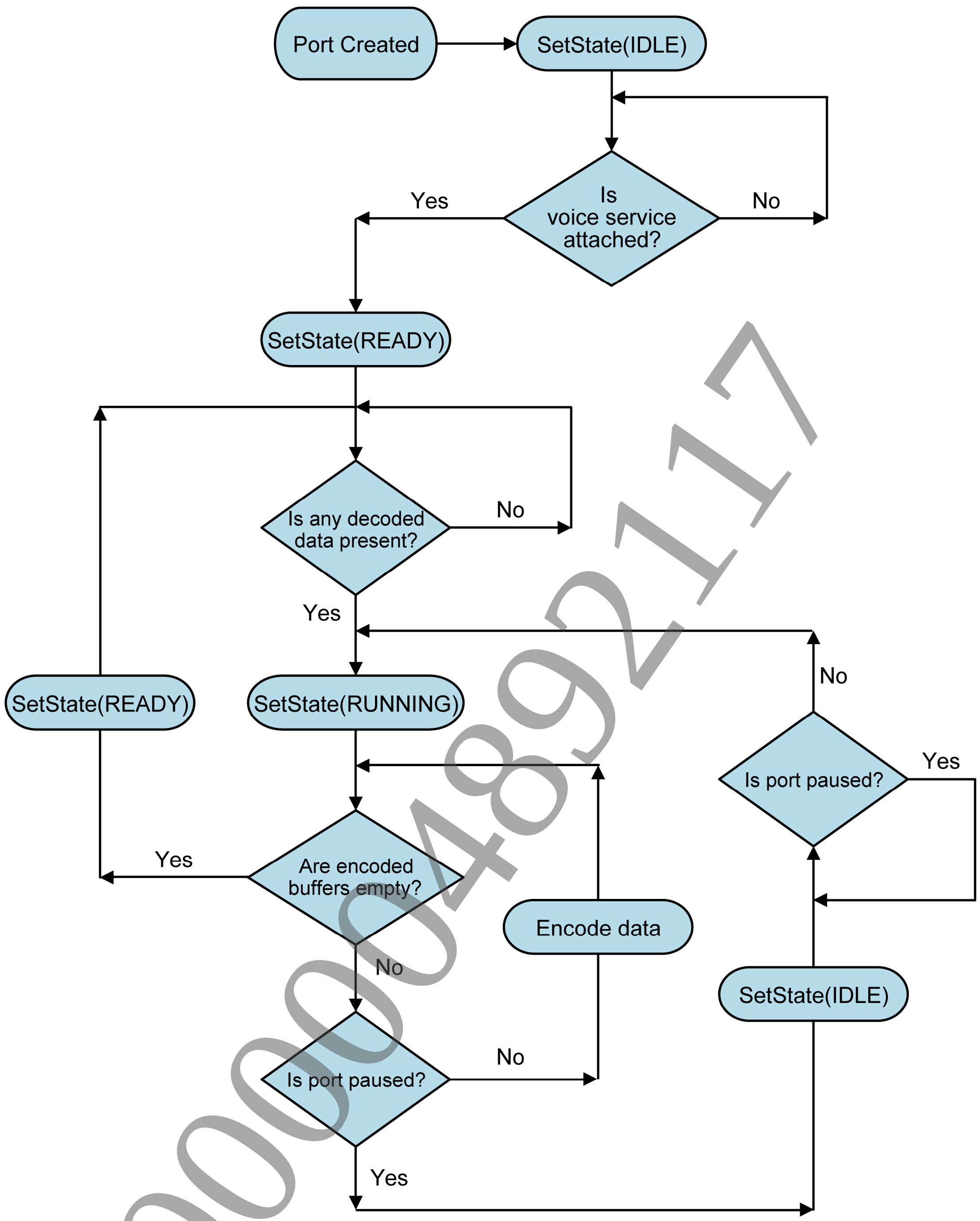


Figure 4 Voice Output Port State



Files

The following files required for using libvoice are shown in Table 1.

Table 1 libvoice Files

File Name	Description
`voice.h`	Header file containing libvoice API functions.
`voice_types.h`	Header file containing data types.
`libSceVoice_stub.a`	Stub library file.

Sample Programs

This section describes two sample programs that use libvoice.

Loop Program

This program is located at:

```
samples/sample_code/audio_video/api_libvoice/loop
```

This program shows the two basic local loopback voice topologies: voice loopback and file loopback. It creates the two connections shown in the following diagram. It then uses polling to get the voice data and, it pumps the data into the corresponding receiving port.

```
Voice Loopback

     Microphone------------------------------> VoiceOutPort
     <voice output port pumps data into voice input port>
     VoiceInPort ----------------------------> HeadsetPort

File Loopback

     PCMAudioInPort -------------------------> HeadsetPort
```

Event Program

This program is located at:

```
samples/sample_code/audio_video/api_libvoice/event
```

This program shows how to receive libvoice events with a callback function. A simple voice-loopback topology is set up as shown in the following diagram:

```
     Microphone------------------------------> VoiceOutPort
     <voice output port pumps data into voice input port>
     VoiceInPort ----------------------------> HeadsetPort
```

The program gets the voice data from the voice output port by using the libvoice event callback scheme. It then pumps the voice data into the voice input port.

2 Using the Library

Basic Procedure

(1) Load PRX Module

Load the PRX module as follows:

```
sceSysmoduleLoadModule(SCE_SYSMODULE_VOICE);
```

(2) Initialize

To use libvoice, call `sceVoiceInit()` to initialize the Voice Service.

```
SceVoiceInitParam params;
   params.appType = SCEVOICE_APPTYPE_GAME;
   params.onEvent = NULL; or // Disable Voice Event System
   params.onEvent = onEvent; // Register callback for receiving Voice Event
   ret = sceVoiceInit(&params, SCEVOICE_VERSION_100);
```

(3) Create IPort and OPort

The application calls `sceVoiceCreatePort()` to create an inbound IPort or an outbound OPort. The function returns the port ID so the application can access the port later.

(4) Connect IPort to OPort

The application calls `sceVoiceConnectIPortToOPort()` to connect IPort to OPort. Repeat step (a) and step (b) to build a voice topology.

Note: An OPort cannot be directly connected to an IPort. See Chapter 3, Topology Use Cases, for detailed port topology setup.

For creating a port that is associated with a specific device, libvoice provides these port-type enumerations:

```
SCEVOICE_PORTTYPE_IN_DEVICE
SCEVOICE_PORTTYPE_OUT_DEVICE
```

Note: `SCEVOICE_PORTTYPE_OUT_DEVICE` corresponds to the receiver stream in the audio service, and the output device depends upon user settings. For details, refer to the *Audio Output Function Overview* document.

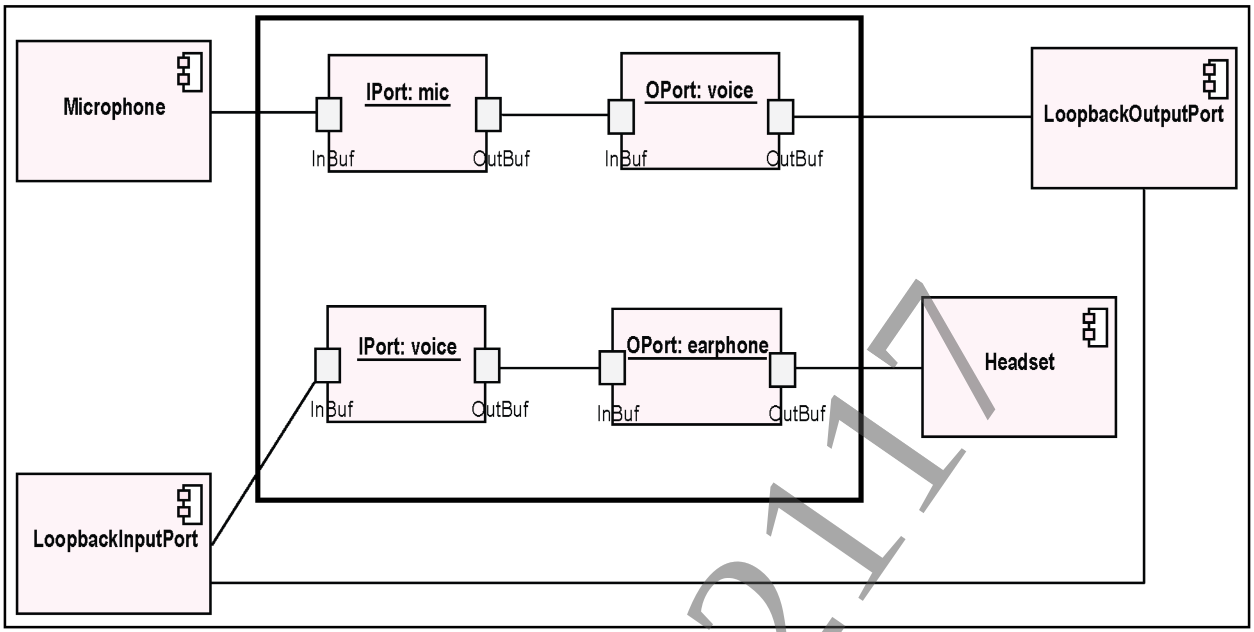
The application must create the libvoice device ports that correspond to the actual devices and set up the topology, but can leave the voice service to take care of all the operations related to a device. The voice service loads the device library, opens the devices, handles the read/write operations, and so on. In other words, the application does not need to deal with the device at all.

If the application wants to deal with the device directly, it can still utilize the voice services. The application can handle the microphone by loading the audioin library and using the `sceAudioIn` APIs. It calls `sceVoiceCreatePort()` to create a `SCEVOICE_PORTTYPE_IN_PCMAUDIO` port. The application then reads the data out of the microphone and calls `sceVoiceWriteToIPort()` to feed the microphone data to the `IN_PCMAUDIO` Port. Do not use `AudioIn` if a `SCEVOICE_PORTTYPE_IN_DEVICE` port has been opened. This produces nondeterministic results.

- PCM data is currently restricted to be 16 KHz and mono. The PCM buffer size has to be less than 32 KB.
- The remaining port types are not device related, and the application must create these ports, connect them to the configuration topology, write to the input ports, and read from the output ports.

Figure 5 shows an example of loopback topology; it is followed by example code for creating and connecting to the illustrated ports

Figure 5 Use Case for Loopback Connection



Make sure all fields in the port are initialized for each specific port.

```
PortArgs.portType = SCEVOICE_PORTTYPE_IN_DEVICE;
PortArgs.bMute = false;
PortArgs.threshold = 100;
PortArgs.volume = 1.0f;
PortArgs.device.playerId = 0;

CallStatus = sceVoiceCreatePort( &MicrophonePort, &PortArgs );

PortArgs.portType = SCEVOICE_PORTTYPE_OUT_DEVICE;
PortArgs.bMute = false;
PortArgs.threshold = 100;
PortArgs.volume = 1.0f;
PortArgs.device.playerId = 0;

CallStatus = sceVoiceCreatePort( &HeadsetPort, &PortArgs );

PortArgs.portType = SCEVOICE_PORTTYPE_OUT_VOICE;
PortArgs.bMute = false;
PortArgs.threshold = 100;
PortArgs.volume = 1.0f;
PortArgs.voice.bitrate = SCEVOICE_BITRATE_7300;
CallStatus = sceVoiceCreatePort( &LoopbackOutputPort, &PortArgs );

PortArgs.portType = SCEVOICE_PORTTYPE_IN_VOICE;
PortArgs.bMute = false;
PortArgs.threshold = 100;
PortArgs.volume = 1.0f;
PortArgs.voice.bitrate = SCEVOICE_BITRATE_7300;
CallStatus = sceVoiceCreatePort( &LoopbackInPort, &PortArgs );

CallStatus = sceVoiceConnectIPortToOPort( MicrophonePort,
                                          LoopbackOutputPort );
CallStatus = sceVoiceConnectIPortToOPort( LoopbackInputPort,
                                          HeadsetPort );
```

(5) Request to Start Voice Service

Call `sceVoiceStart()` to start the Voice Service:

```
SceVoiceStartParam startParams;
startParams.container = sceKernelAllocMemBlock("SceUserTest",
SCE_KERNEL_MEMBLOCK_TYPE_USER_RWDATA, SCE_VOICE_MEMORY_CONTAINER_SIZE,
SCE_NULL);
ret = sceVoiceStart(&startParams);
```

The application can then read and write to the ports to stream voice data after the call returns. The `sceVoiceReadFromOPort()` and `sceVoiceWriteToIPort()` calls return `SCE_VOICE_ERROR_SERVICE_DETACHED` if the voice service is not attached.

(6) Write Stream Data to IPort

The application periodically calls `sceVoiceWriteToIPort()` to feed stream data to the inbound IPorts.

```
CallStatus = sceVoiceWriteToIPort( InputPort, Buffer, &Size, FrameGaps );
```

`Size` specifies how many bytes to write and returns how many bytes were actually written. `FrameGaps` is the number of frames dropped between voice data writings. The *libvoice Reference* describes `sceVoiceWriteToIPort()` in more detail.

(7) Read Stream Data from OPort

The application periodically calls `sceVoiceReadFromOPort()` to retrieve stream data from the outbound OPorts.

The application can also call `sceVoiceReadFromOPort()` while receiving the `SCEVOICE_EVENT_DATA_READY` event at the registered callback function.

```
CallStatus = sceVoiceReadFromOPort( LoopbackOutputPort, LoopbackBuffer,
                                    &Size );
```

`Size` specifies the number of bytes to read and returns the number of bytes actually read. The number of bytes read is always aligned to the individual codec unit data size (19 bytes, 20 bytes, 4 bytes, and so on). In other words, the `sceVoiceReadFromOPort()` always returns an aligned block size. Your application should not randomly split data packets across networks, because the data needs to be aligned to a unit size boundary.

(8) Voluntarily Stop Voice Service

The application can call `sceVoiceStop()` to voluntarily stop the voice service. All active ports go into an idle state (`SCEVOICE_PORTSTATE_IDLE`). The function `sceVoiceStop()` releases some memory, yet keeps the topology information valid.

The application can call `sceVoiceStart()` to start the service again.

(9) Termination

The application calls `sceVoiceEnd()` to exit the voice service, destroy the topology information, and free all used resources. The application must set up everything again when it restarts.

(10) Unload PRX Module

Finally, unload the PRX module using:

```
sceSysmoduleUnloadModule(SCE_SYSMODULE_VOICE);
```

Set/Get Attributes

The libvoice library provides `sceVoiceSetPortAttr()` and `sceVoiceGetPortAttr()` to set and get attributes related to different port types. Currently, only the `SCEVOICE_ATTR_AUDIOINPUT_ACTIVE` attribute is supported (this attribute is only applicable to the `SCEVOICE_PORTTYPE_IN_DEVICE` port type). This attribute is used to get the current AudioInput device status. When the AudioInput device is inactive, the voice data is muted by the AudioInput driver. Note that `sceVoiceSetPortAttr()` has no effect on this attribute.

The AudioInput driver activates only one of the ports that multiple applications are using on the PSP2 system, and it mutes all other ports. The port that will be activated can be selected by system setting and user operation. `sceVoiceGetPortAttr()` can be used to check if the port is activated or not.

Voice Event Callback

The voice event callback function is registered at `sceVoiceInit()` as follows:

```
SceVoiceInitParam params;
params.appType = SCEVOICE_APPTYPE_GAME;
params.onEvent = onEvent;
params.pUserData = 0;   // Arbitary user data pointer
ret = sceVoiceInit(&params, SCEVOICE_VERSION_100);
```

A pointer to an `SceVoiceEventData` structure is then passed to the event callback function. The application processes the event accordingly:

```
void onEvent(SceVoiceEventData * pEvent)
{
    // Go through each SCEVOICE_PORTTYPE_OUT_VOICE port with voice data ready
    if (pEvent->eventType == SCEVOICE_EVENT_DATA_READY)
    {
        For (int i=0; i< pEvent ->dataReady.num; ++i)
        {
            ret = sceVoiceReadFromOPort(pEvent ->dataReady.pid[i], Buffer,
                                        &Size);
        }
    }
}
```

List of Functions

This section lists the functions that libvoice provides. All the functions are blocking calls.

Initialization and Termination (not multithread safe)

Function	Description
`sceVoiceInit()`	Initializes libvoice for system memory.
`sceVoiceEnd()`	Terminates libvoice.

Topology Management (multithread safe)

Function	Description
`sceVoiceCreatePort()`	Creates inbound IPort or outbound OPort.
`sceVoiceConnectIPortToOPort()`	Connects IPort to OPort.
`sceVoiceDisconnectIPortFromOPort()`	Disconnects IPort from OPort.
`sceVoiceDeletePort()`	Deletes inbound IPort or outbound OPort.
`sceVoiceCheckTopology()`	Debug function to verify a streamable topology.

Start/Stop Service (multithread safe)

Function	Description
`sceVoiceStart()`	Starts the Voice Service with system memory space.
`sceVoiceStop()`	Stops the Voice Service.

I/O Access (multithread safe)

Function	Description
`sceVoiceWriteToIPort()`	Writes data to IPort input buffer.
`sceVoiceReadFromOPort()`	Reads data from OPort output buffer.

Update and Query (multithread safe)

Function	Description
`sceVoiceUpdatePort()`	Updates IPort/OPort with the new value of the `SceVoicePortParam` struct.
`sceVoiceSetMuteFlagAll()`	Sets the Mute flag on/off for all ports.
`sceVoiceSetMuteFlag()`	Sets the Mute flag on/off for a specific port.
`sceVoiceSetVolume()`	Sets the volume for a specific port.
`sceVoiceSetBitRate()`	Sets the bit rate value for a specified port.
`sceVoiceSetPortAttr()`	Sets a port attribute through setting `SceVoicePortAttr`.
`sceVoiceGetPortInfo()`	Gets the basic port information for a specified port.
`sceVoiceGetMuteFlag()`	Gets the Mute flag for a specified port.
`sceVoiceGetVolume()`	Gets the volume for a specified port.
`sceVoiceGetBitRate()`	Gets the bit rate value for a specified port.
`sceVoiceGetPortAttr()`	Gets a port attribute setting
`sceVoiceResetPort()`	Resets the port to its initial state.

Pause/Resume Ports (multithread safe)

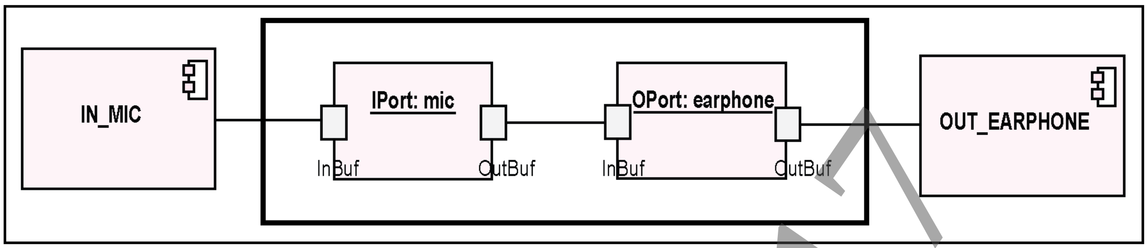
Function	Description
`sceVoicePausePort()`	Pauses streaming of the specified port.
`sceVoicePausePortAll()`	Pauses streaming for all ports.
`sceVoiceResumePort()`	Resumes streaming of the specified the port.
`sceVoiceResumePortAll()`	Resumes streaming for all ports.

3 Topology Use Cases

The following topology cases have been tested for the release.

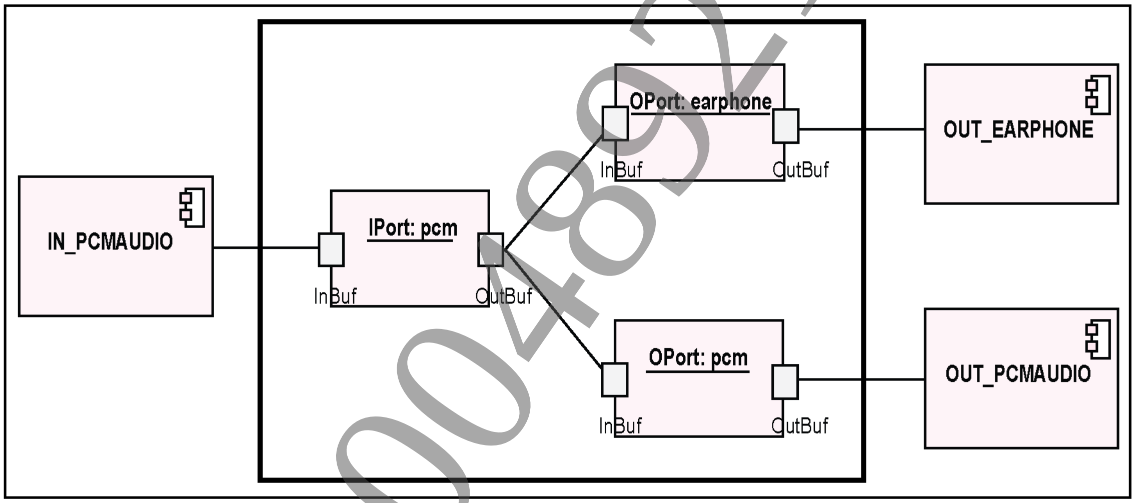
Microphone to Earphone

Figure 6 Use Case for Microphone to Earphone



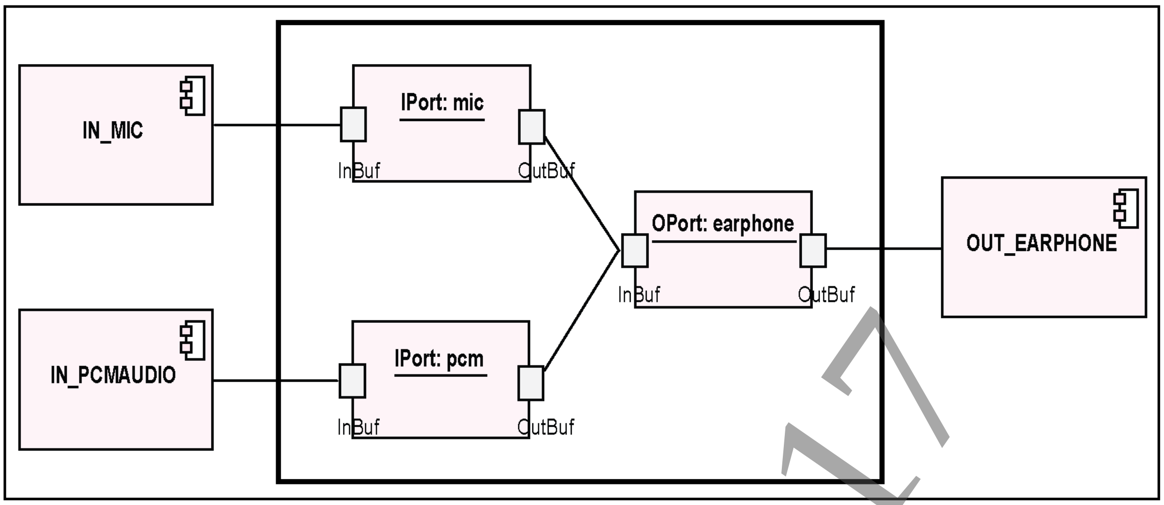
File Playback and Record

Figure 7 Use Case for File Playback and Record



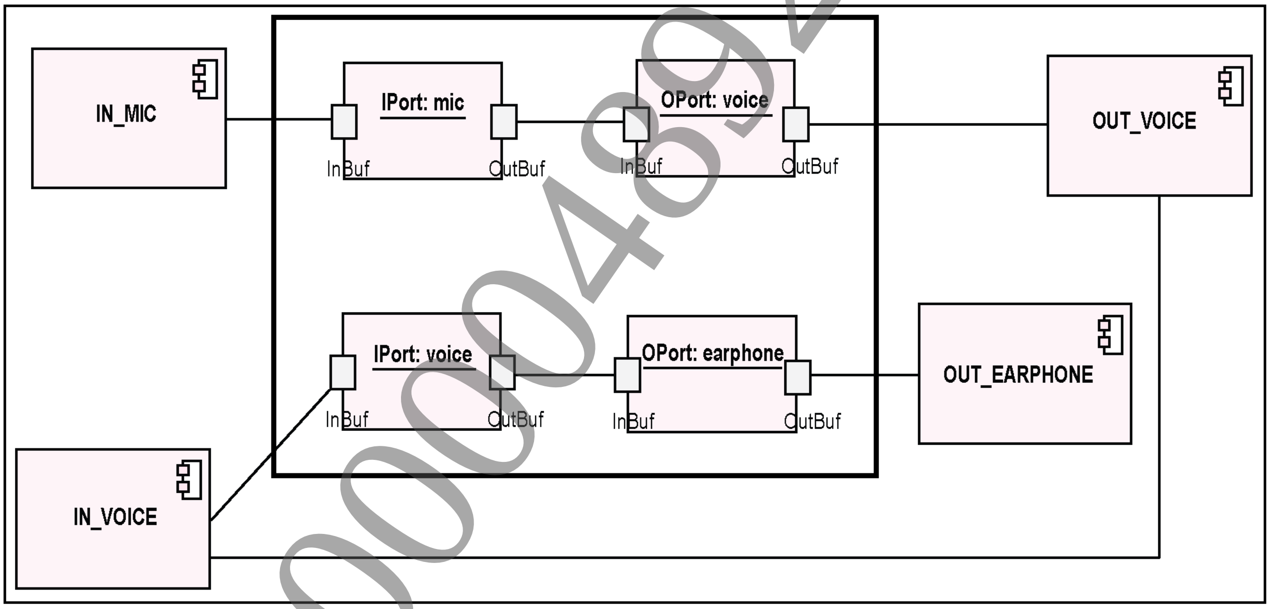
Mixing

Figure 8 Use Case for Mixing



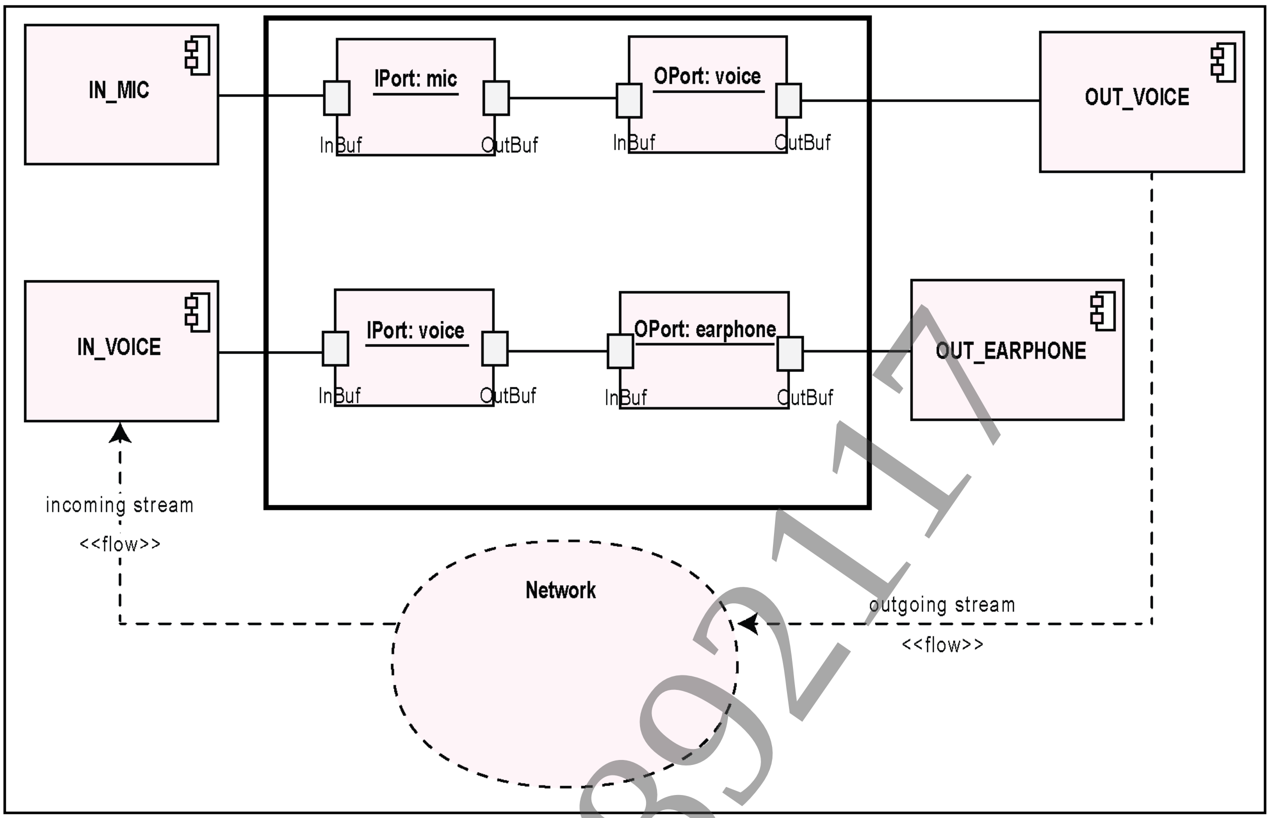
Voice Sending/Receiving

Figure 9 Use Case for Voice Sending/Receiving



Chat over Network

Figure 10 Use Case for Chat over Network


IN_MIC
IPort: mic
InBuf
OutBuf
OPort: voice
InBuf
OutBuf
OUT_VOICE
IN_VOICE
IPort: voice
InBuf
OutBuf
OPort: earphone
InBuf
OutBuf
OUT_EARPHONE
incoming stream
<<flow>>
Network
outgoing stream
<<flow>>