QNX® Neutrino® OS
Audio Developer's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Thursday, May 1, 2014

Table of Contents

About This Guide

The *Audio Developer's Guide* is intended for developers who wish to write audio applications using the QNX Sound Architecture (QSA) drivers and library.

This table may help you find what you need in this guide:

To find out about:	Go to:
The structure of an audio application	*Audio Architecture* (p. 15)
Playing and recording sound	*Playing and Capturing Audio Data* (p. 25)
The structure of a mixer	*Mixer Architecture* (p. 41)
Some tips for reducing audio latency	*Optimizing Audio* (p. 51)
Audio library functions	*Audio Library* (p. 53)
How to code a `.wav` player in C	*`wave.c` example*
How to code a `.wav` recorder in C	*`waverec.c` example*
How to code a `mix_ctl` in C	*`mix_ctl.c` example*
Why `libasound.a` isn't offered	*ALSA and `libasound.so`*
Changes made in each release	*What's New in This Release?*
Terms used in this guide	*Glossary*

You should have already installed the QNX Neutrino RTOS and become familiar with its architecture. For a detailed overview, see the *System Architecture* guide.

The key components of the QNX Audio driver architecture include:

`io-audio`

Audio system manager.

`deva-ctrl-*.so` drivers

Audio drivers. For example, the audio driver for the Ensoniq Audio PCI cards is `deva-ctrl-audiopci.so`. For more information, see the entries for the `deva-*` audio drivers in the *Utilities Reference*.

`libasound.so`

Programmer interface library.

`<asound.h>`, `<asoundlib.h>`

Header files in `/usr/include/sys/`.

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	`Enter`
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Audio Architecture

This chapter includes information about QNX Sound Architecture (QSA), sounds cards and devices, sound card control devices, mixers, and pulse code modulation (PCM).

QNX Sound Architecture

In order for an application to produce sound, the system must include several components.

- hardware in the form of a sound card or sound chip
- a device driver for the hardware
- a well-defined way for the application to talk to the driver, in the form of an Application Programming Interface (API).

This whole system is referred to as the *QNX Sound Architecture (QSA)*. QSA has a rich heritage and owes a large part of its design to version 0.5.2 of the Advanced Linux Sound Architecture (ALSA), but as both systems continued to develop and expand, direct compatibility between the two was lost.

This document concentrates on defining the API and providing examples of how to use it. But before defining the API calls themselves, you need a little background on the architecture itself. If you want to jump in right away, see the examples of a “wav” player and a “wav” recorder in the *wave.c* and *waverec.c* appendixes.

Cards and devices

The basic piece of hardware needed to produce or capture (i.e., record) sound is an audio chip or sound card, referred to simply as a *card*. QSA can support more than one card at a time, and can even mount and unmount cards “on the fly” (more about this later). All the sound devices are attached to a card, so in order to reach a device, you must first know what card it's attached to.

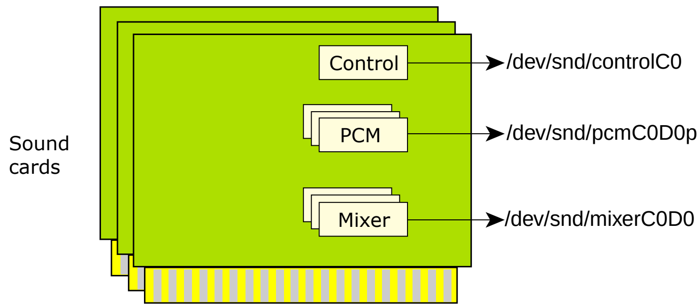

Figure 1: Cards and devices.

The devices include:

- *Control* (p. 18)
- *Mixer* (p. 19)
- *Pulse Code Modulation (PCM)* (p. 20)

You can list the devices that are on your system by typing:

```
ls /dev/snd
```

The resulting list includes one control device for every sound card, starting from card 0, as well as the PCM and mixer devices for each card.

Control device

There's one control device for each sound card in the system.

This device is special because it doesn't directly control any real hardware. It's a concentration point for information about its card and the other devices attached to its card. The primary information kept by the control device includes the type and number of additional devices attached to the card.

Mixer devices

Mixer devices are responsible for combining or mixing the various analog signals on the sound card.

A mixer may also provide a series of controls for selecting which signals are mixed and how they're mixed together, adjusting the gain or attenuation of signals, and/or the muting of signals.

For more information, see the *Mixer Architecture* (p. 41) chapter.

Pulse Code Modulation (PCM) devices

PCM devices are responsible for converting digital sound sequences to analog waveforms, or analog waveforms to digital sound sequences.

Each device operates only in one mode or the other. If it converts digital to analog, it's a *playback* channel device; if it converts analog to digital, it's a *capture* channel device.

The attributes of PCM devices include:

- the data formats that the device supports (16-bit signed little endian, 32-bit unsigned big endian, etc.). For more information, see "*Data formats* (p. 20)," below.
- the data rates that the device can run at (48KHz, 44.1 kHz etc.)
- the number of streams that the device can support (e.g., 2-channel stereo, mono, and 4-channel surround)
- the number of simultaneous clients that the device can support, referred to as the number of *subchannels* the device has. Most sound cards support only 1 subchannel, but some cards can support more; for example, the Soundblaster Live! supports 32 subchannels.

  The maximum number of subchannels supported is a hardware limitation. On single-subchannel cards, this limitation is artificially surpassed through a software solution: the software subchannel mixer. This allows 8 software subchannels to exist on top of the single hardware subchannel.

  The number of subchannels that a device advertises as supporting is defined for the best-case scenario; in the real world, the device might support fewer. For example, a device might support 32 simultaneous clients if they all run at 48 kHz, but might support only 8 clients if the rate is 44.1 kHz. In this case, the device advertises 32 subchannels.

Data formats

The QNX Sound Architecture supports a variety of data formats.

The `<asound.h>` header file defines two sets of constants for the data formats. The two sets are related (and easily converted between) but serve different purposes:

SND_PCM_SFMT_*

A single selection from the set of data formats. For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211) in the Audio Library chapter.

SND_PCM_FMT_*

A group of (one or more) formats within a single variable. This is useful for specifying the format capabilities of a device, for example.

Generally, the SND_PCM_FMT_* constants are used to convey information about raw potential, and the SND_PCM_SFMT_* constants are used to select and report a specific configuration.

You can build a format from its width and other attributes, by calling *snd_pcm_build_linear_format()* (p. 143).

You can use these functions to check the characteristics of a format:

- *snd_pcm_format_big_endian()* (p. 194)
- *snd_pcm_format_linear()* (p. 196)
- *snd_pcm_format_little_endian()* (p. 198)
- *snd_pcm_format_signed()* (p. 200)
- *snd_pcm_format_unsigned()* (p. 205)

PCM state machine

A PCM device is, at its simplest, a data buffer that's converted, one sample at a time, by either a Digital Analog Converter (DAC) or an Analog Digital Converter (ADC), depending on direction.

This simple idea becomes a little more complicated in QSA because of the concept that the PCM subchannel is in a state at any given moment. These states are defined as follows:

SND_PCM_STATUS_NOTREADY

The initial state of the device.

SND_PCM_STATUS_READY

The device has its parameters set for the data it will operate on.

SND_PCM_STATUS_PREPARED

The device has been prepared for operation and is able to run.

SND_PCM_STATUS_RUNNING

The device is running, transferring data to or from the buffer.

SND_PCM_STATUS_UNDERRUN

This state happens only to a playback device and is entered when the buffer has no more data to be played.

SND_PCM_STATUS_OVERRUN

This state happens only to a capture device and is entered when the buffer has no room for data.

SND_PCM_STATUS_PAUSED

The device has been paused.

SND_PCM_STATUS_UNSECURE

The application marked the stream as protected, the hardware level supports a secure transport (e.g., HDCP for HDMI), and authentication was lost.

SND_PCM_STATUS_ERROR

A hardware error has occurred, and the stream must be prepared again.

SND_PCM_STATUS_CHANGE

The stream has changed and must be prepared again.

In some cases, audio is redirected transparently between different audio devices, with different capabilities. You can enable this by calling:

```
snd_pcm_plugin_set_enable(handle, PLUGIN_ROUTING);
```

If routing is enabled and the preferred device changes to an external device such as HDMI, audio is automatically redirected to that device. Once routing changes, the client receives a status of SND_PCM_STATUS_CHANGE.

When the client receives this, it may just call *snd_pcm_channel_prepare()* (p. 173) to enter the SND_PCM_STATUS_PREPARED state, and be ready to continue playback, but as the capabilities may have changed, (e.g., HDMI may support surround sound output while a local speaker doesn't), the client may wish to call *snd_pcm_channel_info()* (p. 159) to check the capabilities, and change the audio characteristics in response before preparing.

SND_PCM_STATUS_PREEMPTED

Audio is blocked because another `libasound` session has initiated playback, and the audio driver has determined that that session has higher priority, and therefore the lower priority session is terminated with state SND_PCM_STATUS_PREEMPTED. When it receives this state, the client should give up on playback, and not attempt to resume until either the sound it wishes to produce has increased in priority, or a user initiates a retry.

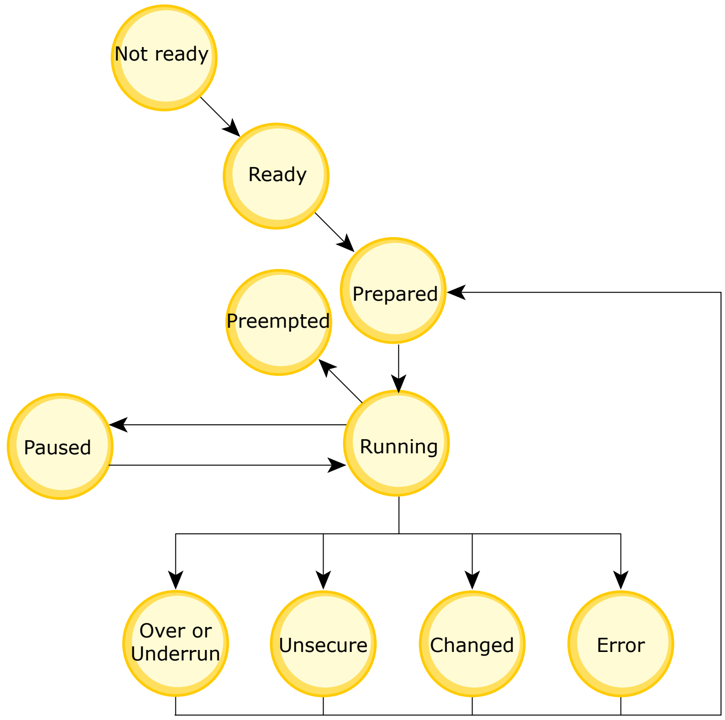

Figure 2: General state diagram for PCM devices.

The transition between states is the result of executing an API call, or the result of conditions that occur in the hardware. For more details, see the *Playing and Capturing Audio Data* (p. 25) chapter.

Software PCM mixing

In the case where the sound card has a playback PCM device with only one subchannel, the device driver writer can choose to include a PCM software mixing device.

This device simply appears as a new PCM playback device that supports many subchannels, but it has a few differences from a true hardware device:

- The mixing of the PCM streams is done in software using the CPU. Even with only one stream, the CPU is used more than if the hardware device were used.
- When the PCM software mixer is started, it opens a connection to the real hardware device. If the real hardware device is already in use, the PCM software mixer can't run. Likewise, if the PCM software mixer is running, the real hardware device is in use and is unavailable.

The PCM software mixer is specifically attached to a single hardware PCM device. This one-to-one mapping allows for an API call to identify the PCM software-mixing device associated with its hardware device.

PCM plugin converters

In some cases, an application has data in one form, and the PCM device is capable of accepting data only in another format. Clearly this won't work unless something is done. The application — like some MPG decoders — could reformat its data "on the fly" to a format that the device accepts. Alternatively, the application can ask QSA to do the conversion for it.

The conversation is accomplished by invoking a series of *plugin converters*, each capable of doing a very specific job. As an example, the rate converter converts a stream from one sampling frequency to another. There are plugin converters for bit conversions (8-to-16-bit, etc.), endian conversion (little endian to big endian and vice versa), voice conversions (stereo to mono, etc.) and so on.

The minimum number of converters is invoked to translate the input format to the output format so as to minimize CPU usage. An application signals its willingness to use the plugin converter interface by using the PCM plugin API functions. These API functions all have `plugin` in their names. For more information, see the *Audio Library* (p. 53) chapter.

The ability to convert audio to match hardware capabilities (for example, voice conversion, rate conversion, type conversion, etc.) is enabled by default. This impacts the functions *snd_pcm_channel_params()* (p. 165), *snd_pcm_channel_setup()* (p. 177), and *snd_pcm_channel_status()* (p. 182). These behave as *snd_pcm_plugin_params()* (p. 247), *snd_pcm_plugin_setup()* (p. 266), and *snd_pcm_plugin_status()* (p. 270), unless you've disabled the conversion by calling:

```
snd_pcm_plugin_set_disable(handle, PLUGIN_CONVERSION);
```

Don't mix the plugin API functions with the nonplugin functions.

Chapter 2
Playing and Capturing Audio Data

This chapter describes the major steps required to play back and capture (i.e., record) sound data.

Handling PCM devices

The software processes for playing back and capturing audio data are similar. This section describes the common steps.

Opening your PCM device

The first thing you need to do in order to playback or capture sound is open a connection to a PCM playback or capture device.

The API calls for opening a PCM device are:

snd_pcm_open_name() **(p. 223)**

Use this call when you want to open a specific hardware device, and you know its name.

snd_pcm_open() **(p. 221)**

Use this call when you want to open a specific hardware device, and you know its card and device number.

snd_pcm_open_preferred() **(p. 226)**

Use this call to open the user's preferred device.

Using this function makes your application more flexible, because you don't need to know the card and device numbers; the function can pass back to you the card and device that it opened.

All these API calls set a PCM connection handle that you'll use as an argument to all other PCM API calls. This handle is very analogous to a file stream handle. It's a pointer to a `snd_pcm_t` structure, which is an opaque data type.

These functions, like others in the QSA API, work for both capture and playback channels. They take as an argument a *channel direction*, which is one of:

- SND_PCM_OPEN_CAPTURE
- SND_PCM_OPEN_PLAYBACK

This code fragment from the *wave.c example* in the appendix uses these functions to open a playback device:

```
if (card == -1)
{
    if ((rtn = snd_pcm_open_preferred (&pcm_handle,
                  &card, &dev,
                  SND_PCM_OPEN_PLAYBACK)) < 0)
        return err ("device open");
}
else
{
    if ((rtn = snd_pcm_open (&pcm_handle, card, dev,
```

```
                 SND_PCM_OPEN_PLAYBACK)) < 0)
        return err ("device open");
}
```

If the user specifies a card and a device number on the command line, this code opens a connection to that specific PCM playback device. If the user doesn't specify a card, the code creates a connection to the preferred PCM playback device, and *snd_pcm_open_preferred()* stores the card and device numbers in the given variables.

Configuring the PCM device

The next step in playing back or capturing the sound stream is to inform the device of the format of the data that you're about to send it or want to receive from it.

You can do this by filling in a `snd_pcm_channel_params_t` (p. 167) structure, and then calling *snd_pcm_channel_params()* (p. 165) or *snd_pcm_plugin_params()* (p. 247). The difference between the functions is that the second one uses the plugin converters (see "*PCM plugin converters* (p. 23)" in the Audio Architecture chapter) if required.

If the device can't support the data parameters you're setting, or if all the subchannels of the device are currently in use, both of these functions fail.

The API calls for determining the current capabilities of a PCM device are:

***snd_pcm_plugin_info()* (p. 245)**

Use the plugin converters. If the hardware has a free subchannel, the capabilities returned are extensive because the plugin converters make any necessary conversion.

***snd_pcm_channel_info()* (p. 159)**

Access the hardware directly. This function returns only what the hardware capabilities are.

Both of these functions take as an argument a pointer to a `snd_pcm_channel_info_t` (p. 161) structure. You must set the *channel* member of this structure to the desired direction (`SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`) *before* calling the functions. The functions fill in the other members of the structure.

It's the act of configuring the channel that allocates a subchannel to the client. Stated another way, hundreds of clients can open a handle to a PCM device with only one subchannel, but only one can configure it. After a client allocates a subchannel, it isn't returned to the free pool until the handle is closed. One result of this mechanism is that, from moment to moment, the capabilities of a PCM device change as other applications allocate and free subchannels. Additionally the act of configuring / allocating a subchannel changes its state from `SND_PCM_STATUS_NOTREADY` to `SND_PCM_STATUS_READY`.

If the API call succeeds, all parameters specified are accepted and are guaranteed to be in effect, except for the *frag_size* parameter, which is only a suggestion to the hardware. The hardware may adjust the fragment size, based on hardware requirements. For example, if the hardware can't deal with fragments crossing 64-kilobyte boundaries, and the suggested *frag_size* is 60 kilobytes, the driver will probably adjust it to 64 kilobytes.

Another aspect of configuration is determining how big to make the hardware buffer. This determines how much latency that the application has when sending data to the driver or reading data from it. The hardware buffer size is determined by multiplying the *frag_size* by the *max_frags* parameter, so for the application to know the buffer size, it must determine the actual *frag_size* that the driver is using.

You can do this by calling *snd_pcm_channel_setup()* (p. 177) or *snd_pcm_plugin_setup()* (p. 266), depending on whether or not your application is using the plugin converters. Both of these functions take as an argument a pointer to a `snd_pcm_channel_setup_t` (p. 179) structure that they fill with information about how the channel is configured, including the true *frag_size*.

Controlling voice conversion

The `libasound` library supports devices with up to eight voices.

Configuration of the `libasound` library is based on the maximum number of voices supported in hardware. If the numbers of source and destination voices are different, then *snd_pcm_plugin_params()* (p. 247) instantiates a voice converter.

The default voice conversion behavior is as follows:

From	To	Conversion
Mono	Stereo	Replicate channel 1 (left) to channel 2 (right)
Stereo	Mono	Remove channel 2 (right)
Mono	4-channel	Replicate channel 1 to all other channels
Stereo	4-channel	Replicate channel 1 (front left) to channel 3 (rear left), and channel 2 (front right) to channel 4 (rear right)

Previous versions of `libasound` converted stereo to mono by averaging the left and right channels to generate the mono stream. Now by default, the right channel is simply dropped.

You can use the voice conversion API to configure the conversion behavior and place any source channel in any destination channel slot:

snd_pcm_plugin_get_voice_conversion() **(p. 243)**

Get the current voice conversion structure for a channel

snd_pcm_plugin_set_voice_conversion() **(p. 264)**

Set the current voice conversion structure for a channel

The actual conversion is controlled by the `snd_pcm_voice_conversion_t` structure, which is defined as follows:

```
typedef struct snd_pcm_voice_conversion
{
   uint32_t     app_voices;
   uint32_t     hw_voices;
   uint32_t     matrix[32];
} snd_pcm_voice_conversion_t
```

The matrix member forms a 32-by-32-bit array that specifies how to convert the voices. The array is ranked with rows representing application voices, voice 0 first; the columns represent hardware voices, with the low voice being LSB-aligned and increasing right to left.

For example, consider a mono application stream directed to a 4-voice hardware device. A bit array of:

```
matrix[0] = 0x1;  //  00000001
```

causes the sound to be output on only the first hardware channel. A bit array of:

```
matrix[0] = 0x9;   // 00001001
```

causes the sound to appear on the first and last hardware channel.

Another example would be a stereo application stream to a 6 channel (5.1) output device. A bit array of:

```
matrix[0] = 0x1;  //  00000001
matrix[1] = 0x2;  //  00000010
```

causes the sound to appear on only the front two channels, while:

```
matrix[0] = 0x5;  //  00000101
matrix[1] = 0x2;  //  00000010
```

causes the stream signal to appear on the first four channels (likely the front and rear pairs, but not on the center or LFE channels). The bitmap used to describe the hardware (i.e., the columns) depends on the hardware, and you need to be mindful of the actual hardware you'll be running on to properly map the channels. For example:

- If the hardware orders the channels such that the center channel is the third channel, then bit 2 represents the center.
- If the hardware orders the channels such that the Rear Left is the third channel, then bit 2 represents the Rear Left.

If the number of source voices matches the number of destination voices, the converter isn't invoked, so you won't be able to reroute the channels. If you're playing a stereo file on stereo hardware, you can't use the voice matrix to swap the channels because the voice converter isn't used in this case.

If you call *snd_pcm_plugin_get_voice_conversion()* or *snd_pcm_plugin_set_voice_conversion()* before the voice conversion plugin has been instantiated, the functions fail and return `-ENOENT`.

Preparing the PCM subchannel

The next step in playing back or capturing the sound stream is to prepare the allocated subchannel to run.

Call one of the following functions to prepare the allocated subchannel:

- *snd_pcm_plugin_prepare()* (p. 251) if you're using the plugin interface
- *snd_pcm_channel_prepare()* (p. 173), *snd_pcm_capture_prepare()* (p. 151), or *snd_pcm_playback_prepare()* (p. 237) if you aren't

  The *snd_pcm_channel_prepare()* function simply calls *snd_pcm_capture_prepare()* or *snd_pcm_playback_prepare()*, depending on the channel direction that you specify.

This step and the `SND_PCM_STATUS_PREPARED` state may seem unnecessary, but they're required to correctly handle underrun conditions when playing back, and overrun conditions when capturing. For more information, see "*If the PCM subchannel stops during playback* (p. 33)" and "*If the PCM subchannel stops during capture* (p. 38)," later in this chapter.

Closing the PCM subchannel

When you've finished playing back or capturing audio data, you can close the subchannel by calling *snd_pcm_close()*.

The call to *snd_pcm_close()* (p. 188) releases the subchannel and closes the handle.

Playing audio data

Once you've opened and configured a PCM playback device and prepared the PCM subchannel, you're ready to play back sound data.

There's a complete example of playback in the *wave.c example* in the appendix. You may wish to compile and run the application now, and refer to the running code as you progress through this section.

> If your application has the option to produce playback data in multiple formats, choosing a format that the hardware supports directly will reduce the CPU requirements.

Playback states

Let's consider the state transitions for a PCM device during playback.

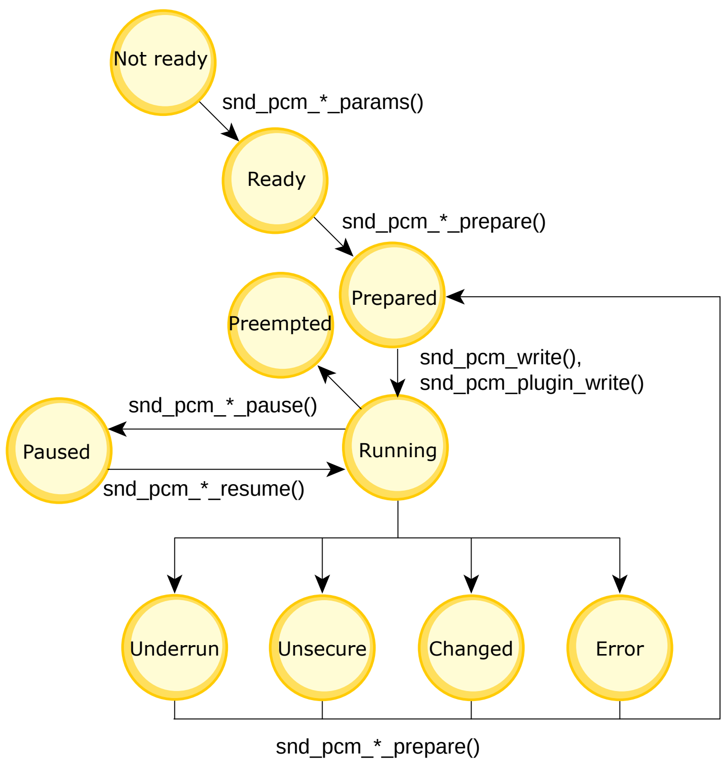

Figure 3: State diagram for PCM devices during playback.

The transition between `SND_PCM_STATUS_*` states is the result of executing an API call, or the result of conditions that occur in the hardware:

From	To	Cause
NOTREADY	READY	Calling *snd_pcm_channel_params()* (p. 165) or *snd_pcm_plugin_params()* (p. 247)
READY	PREPARED	Calling *snd_pcm_channel_prepare()* (p. 173), *snd_pcm_playback_prepare()* (p. 237), or *snd_pcm_plugin_prepare()* (p. 251)
PREPARED	RUNNING	Calling *snd_pcm_write()* (p. 283) or *snd_pcm_plugin_write()* (p. 274)
RUNNING	PAUSED	Calling *snd_pcm_channel_pause()* (p. 171) or *snd_pcm_playback_pause()* (p. 235)
PAUSED	RUNNING	Calling *snd_pcm_channel_resume()* (p. 175) or *snd_pcm_playback_resume()* (p. 239)
RUNNING	UNDERRUN	The hardware buffer became empty during playback
RUNNING	UNSECURE	The application marked the stream as protected, the hardware level supports a secure transport (e.g., HDCP for HDMI), and authentication was lost
RUNNING	CHANGE	The stream changed
RUNNING	ERROR	A hardware error occurred
UNDERRUN, UNSECURE, CHANGE, or ERROR	PREPARED	Calling *snd_pcm_channel_prepare()* (p. 173), *snd_pcm_playback_prepare()* (p. 237), or *snd_pcm_plugin_prepare()* (p. 251)
RUNNING	PREEMPTED	Audio is blocked because another `libasound` session has initiated playback, and the audio driver has determined that that session has higher priority

For more details on these transitions, see the description of each function in the *Audio Library* (p. 53) chapter.

Sending data to the PCM subchannel

The function that you call to send data to the subchannel depends on whether or not you're using plugin converters.

***snd_pcm_write()* (p. 283)**

The number of bytes written must be a multiple of the fragment size, or the write will fail.

***snd_pcm_plugin_write()* (p. 274)**

The plugin accumulates partial writes until a complete fragment can be sent to the driver.

A full nonblocking write mode is supported if the application can't afford to be blocked on the PCM subchannel. You can enable nonblocking mode when you open the handle or by calling *snd_pcm_nonblock_mode()* (p. 219).

This approach results in a polled operation mode that isn't recommended.

Another method that your application can use to avoid blocking on the write is to call *select()* (see the QNX Neutrino *C Library Reference*) to wait until the PCM subchannel can accept more data. This is the technique that the `wave.c` *example* uses. It allows the program to wait on user input while at the same time sending the playback data to the PCM subchannel.

To get the file descriptor to pass to *select()*, call *snd_pcm_file_descriptor()* (p. 190).

With this technique, *select()* returns when there's space for *frag_size* bytes in the subchannel. If your application tries to write more data than this, it may block on the call.

If the PCM subchannel stops during playback

When playing back, the PCM subchannel stops if the hardware consumes all the data in its buffer.

This can happen if the application can't produce data at the rate that the hardware is consuming data. A real-world example of this is when the application is preempted for a period of time by a higher-priority process. If this preemption continues long enough, all data in the buffer may be played before the application can add any more.

When this happens, the subchannel changes state to `SND_PCM_STATUS_UNDERRUN`. In this state, it doesn't accept any more data (i.e., *snd_pcm_write()* (p. 283) and *snd_pcm_plugin_write()* (p. 274) fail) and the subchannel doesn't restart playing.

The only ways to move out of this state are to close the subchannel or to reprepare the channel as you did before (see "*Preparing the PCM subchannel* (p. 30)," earlier in this chapter). This forces the application to recognize and take action to get out of the underrun state; this is primarily for applications that want to synchronize audio with something else. Consider the difficulties involved with synchronization if the subchannel simply were to move back to the `SND_PCM_STATUS_RUNNING` state from underrun when more data became available.

Stopping the playback

If the application wishes to stop playback, it can simply stop sending data and let the subchannel underrun as described above, but there are better ways.

If you want your application to stop as soon as possible, call one of the drain functions to remove any unplayed data from the hardware buffer:

- *snd_pcm_plugin_playback_drain()* (p. 249) if you're using the plugins
- *snd_pcm_playback_drain()* (p. 229) if you aren't

If you want to play out all data in the buffers before stopping, call one of:

- *snd_pcm_plugin_flush()* (p. 241) if you're using the plugins
- *snd_pcm_channel_flush()* (p. 155) or *snd_pcm_playback_flush()* (p. 231) if you aren't

Synchronizing with the PCM subchannel

QSA provides some basic synchronization capabilities.

Your application can find out where in the stream the hardware play position is. The resolution of this position is entirely a function of the hardware driver; consult the specific device driver documentation for details if this is important to your application.

The API calls to get this information are:

- *snd_pcm_plugin_status()* (p. 270) if you're using the plugin interface
- *snd_pcm_channel_status()* (p. 182) if you aren't

Both of these functions fill in a `snd_pcm_channel_status_t` (p. 184) structure. You'll need to check the following members of this structure:

scount

The hardware play position, in bytes relative to the start of the stream since the last time the channel was prepared. The act of preparing a channel resets this count.

count

The play position, in bytes relative to the total number of bytes written to the device.

The *count* member isn't used if the mmap plugin is used. To disable the mmap plugin, call *snd_pcm_plugin_set_disable()* (p. 256).

For example, consider a stream where 1,000,000 bytes have been written to the device. If the status call sets *scount* to 999,000 and *count* to 1000, there are 1000 bytes of data in the buffer remaining to be played, and 999,000 bytes of the stream have already been played.

Capturing audio data

Once you've opened and configured a PCM capture device and prepared the PCM subchannel, you're ready to capture sound data.

For more information about this preparation, see "*Handling PCM devices* (p. 26)."

There's a complete example of capturing audio data in the `waverec.c` *example* in the appendix. You may wish to compile and run the application now, and refer to the running code as you progress through this section.

This section includes:

Selecting what to capture

Most sound cards allow only one analog signal to be connected to the ADC. Therefore, in order to capture audio data, the user or application must select the appropriate input source.

Some sound cards allow multiple signals to be connected to the ADC; in this case, make sure the appropriate signal is one of them. There's an API call, *snd_mixer_group_write()* (p. 116), for controlling the mixer so that the application can set this up directly; it's described in the *Mixer Architecture* (p. 41) chapter.

Capture states

Let's consider the state transitions for PCM devices during capture.

The state diagram for a PCM device during capture is shown below.

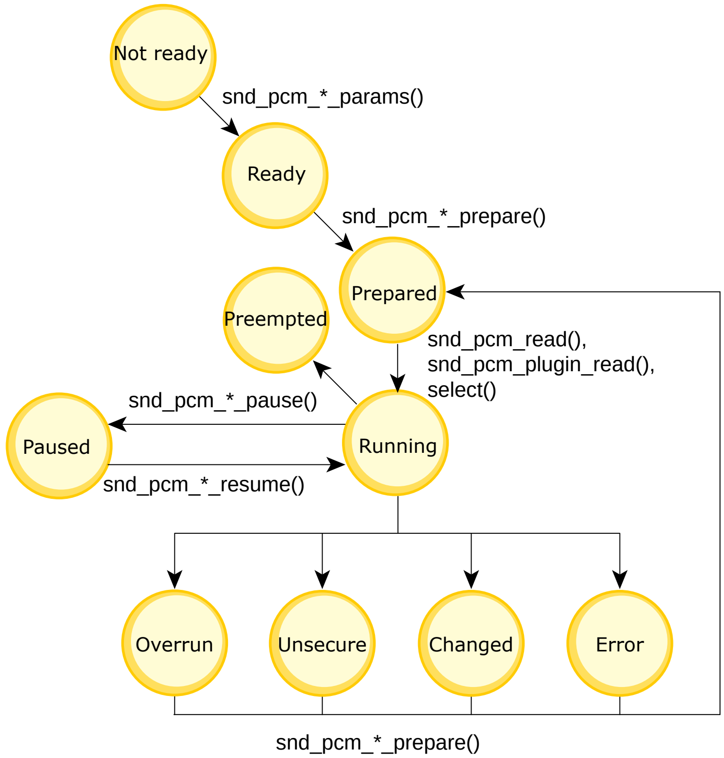

Figure 4: State diagram for PCM devices during capture.

The transition between SND_PCM_STATUS_* states is the result of executing an API call, or the result of conditions that occur in the hardware:

From	To	Cause
NOTREADY	READY	Calling *snd_pcm_channel_params()* (p. 165) or *snd_pcm_plugin_params()* (p. 247)
READY	PREPARED	Calling *snd_pcm_capture_prepare()* (p. 151), *snd_pcm_channel_prepare()* (p. 173), or *snd_pcm_plugin_prepare()* (p. 251)
PREPARED	RUNNING	Calling *snd_pcm_read()* (p. 277) or *snd_pcm_plugin_read()* (p. 253), or calling *select()* against the capture file descriptors
RUNNING	PAUSED	Calling *snd_pcm_capture_pause()* (p. 149) or *snd_pcm_channel_pause()* (p. 171)
PAUSED	RUNNING	Calling *snd_pcm_capture_resume()* (p. 153) or *snd_pcm_channel_resume()* (p. 175)
RUNNING	OVERRUN	The hardware buffer became full during capture; *snd_pcm_read()* (p. 277) and *snd_pcm_plugin_read()* (p. 253) fail

From	To	Cause
`RUNNING`	`UNSECURE`	The application marked the stream as protected, the hardware level supports a secure transport (e.g., HDCP for HDMI), and authentication was lost
`RUNNING`	`CHANGE`	The stream changed
`RUNNING`	`ERROR`	A hardware error occurred
`OVERRUN`, `UNSECURE`, `CHANGE`, or `ERROR`	`PREPARED`	Calling *snd_pcm_capture_prepare()* (p. 151), *snd_pcm_channel_prepare()* (p. 173), or *snd_pcm_plugin_prepare()* (p. 251)
`RUNNING`	`PREEMPTED`	Audio is blocked because another `libasound` session has initiated playback, and the audio driver has determined that that session has higher priority

For more details on these transitions, see the description of each function in the *Audio Library* (p. 53) chapter.

Receiving data from the PCM subchannel

The function that you call to receive data from the subchannel depends on whether or not you're using plugin converters.

snd_pcm_read() **(p. 277)**

The number of bytes read must be a multiple of the fragment size, or the read fails.

snd_pcm_plugin_read() **(p. 253)**

The plugin reads an entire fragment from the driver and then fulfills requests for partial reads from that buffer until another full fragment has to be read.

A full nonblocking read mode is supported if the application can't afford to be blocked on the PCM subchannel. You can enable nonblocking mode when you open the handle or by using the *snd_pcm_nonblock_mode()* (p. 219) API call.

This approach results in a polled operation mode that isn't recommended.

Another method that your application can use to avoid blocking on the read is to use *select()* (see the QNX Neutrino *C Library Reference*) to wait until the PCM subchannel has more data. This is the technique that the `waverec.c` *example* uses. It allows the program to wait on user input while at the same time receiving the capture data from the PCM subchannel.

To get the file descriptor to pass to *select()*, call *snd_pcm_file_descriptor()* (p. 190).

With this technique, *select()* returns when there are *frag_size* bytes in the subchannel. If your application tries to read more data than this, it may block on the call.

If the PCM subchannel stops during capture

When capturing, the PCM subchannel stops if the hardware has no room for additional data left in its buffer.

This can happen if the application can't consume data at the rate that the hardware is producing data. A real-world example of this is when the application is preempted for a period of time by a higher-priority process. If this preemption continues long enough, the data buffer may be filled before the application can remove any data.

When this happens, the subchannel changes state to `SND_PCM_STATUS_OVERRUN`. In this state, it won't provide any more data (i.e., *snd_pcm_read()* (p. 277) and *snd_pcm_plugin_read()* (p. 253) fail) and the subchannel doesn't restart capturing.

The only ways to move out of this state are to close the subchannel or to reprepare the channel as you did before. This forces the application to recognize and take action to get out of the overrun state; this is primarily for applications that want to synchronize audio with something else. Consider the difficulties involved with synchronization if the subchannel simply were to move back to the `SND_PCM_STATUS_RUNNING` state from overrun when space became available; the recorded sample would be discontinuous.

Stopping the capture

If your application wishes to stop capturing, it can simply stop reading data and let the subchannel overrun as described above, but there's a better way.

If you want your application to stop capturing immediately and delete any unread data from the hardware buffer, call one the flush functions:

- *snd_pcm_plugin_flush()* (p. 241) if you're using the plugins
- *snd_pcm_channel_flush()* (p. 155) or *snd_pcm_capture_flush()* (p. 145) if you aren't

Synchronizing with the PCM subchannel

QSA provides some basic synchronization capabilities.

An application can find out where in the stream the hardware capture position is. The resolution of this position is entirely a function of the hardware driver; consult the specific device driver documentation for details if this is important to your application.

The API calls to get this information are:

- *snd_pcm_plugin_status()* (p. 270) if you're using the plugin interface
- *snd_pcm_channel_status()* (p. 182) if you aren't

Both of these functions fill in a `snd_pcm_channel_status_t` (p. 184) structure. You'll need to check the following members of this structure:

scount

The hardware capture position, in bytes relative to the start of the stream since you last prepared the channel. The act of preparing a channel resets this count.

count

The capture position as bytes in the hardware buffer.

The *count* member isn't used if the mmap plugin is used. To disable the mmap plugin, call *snd_pcm_plugin_set_disable()* (p. 256).

Chapter 3
Mixer Architecture

This section describes the mixer architecture.

You can usually build an audio mixer from a relatively small number of components. Each of these components performs a specific mixing function. A summary of these components or *elements* follows:

Input

A connection point where an external analog signal is brought into the mixer.

Output

A connection point where an analog signal is taken from the mixer.

ADC

An element that converts analog signals to digital samples.

DAC

An element that converts digital samples to analog signals.

Switch

An element that can connect two or more points together. A simple switch may be used as a mute control. More complicated switches can mute the channels of a stream individually, or can even form crossbar matrices where *n* input signals can be connected to *n* output signals.

Volume

An element that adjusts the amplitude level of a signal by applying attenuation or gain.

Accumulator

An element the adds all signals input to it and produces an output signal.

Multiplexer

An element that selects the signal from one of its inputs and forwards it to a single output line.

By using these elements you can build a simple sound card mixer:

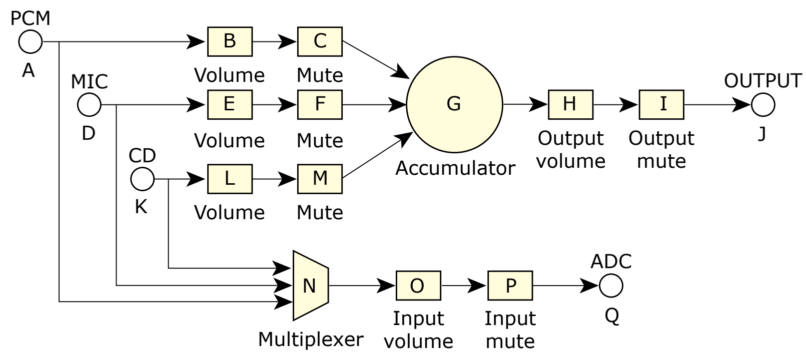

Figure 5: A simple sound card mixer.

In the diagram, the mute figures are switches, and the MIC and CD are input elements. This diagram is in fact a simplified representation of the Audio Codec '97 mixer, one of the most common mixers found on sound cards.

It's possible to control these mixer elements directly using the *snd_mixer_element_read()* (p. 93) and *snd_mixer_element_write()* (p. 96) functions, but this method isn't recommended because:

- The arguments to these functions are very dependent on the element type.
- Controlling many elements to change mixer functionality is difficult with this method.
- There's a better method.

The element interface is the lowest level of control for a mixer and is complicated to control. One solution to this complexity is to arrange elements that are associated with a function into a *mixer group*. To further refine this idea, groups are classified as either playback or capture groups. To simplify creating and managing groups, a hard set of rules was developed for how groups are built from elements:

- A *playback group* contains at most one volume element and one switch element (as a mute).
- A *capture group* contains at most one each of a volume element, switch element (as a mute), and capture selection element. The capture selection element may be a multiplexer or a switch.

If you apply these rules to the simple mixer in the above diagram, you get the following:

Playback Group PCM

Elements *B* (volume) and *C* (switch).

Playback Group MIC

Elements *E* (volume) and *F* (switch).

Playback Group CD

Elements *L* (volume) and *M* (switch).

Playback Group MASTER

Elements *H* (volume) and *I* (switch).

Capture Group MIC

Element *N* (multiplexer); there's no volume or switch.

Capture Group CD

Element *N* (multiplexer); there's no volume or switch.

Capture Group INPUT

Elements *O* (volume) and *P* (switch).

In separating the elements into groups, you've reduced the complexity of control (there are 7 groups instead of 17 elements), and each group associates well with what applications want to control.

Opening the mixer device

To open a connection to the mixer device, call *snd_mixer_open()*.

This call has arguments for selecting the card and mixer device number to open. Most sound cards have only one mixer, but there may be additional mixers in special cases.

The *snd_mixer_open()* (p. 124) call returns a mixer handle that you'll use as an argument for additional API calls applied to this device. It's a pointer to a `snd_mixer_t` structure, which is an opaque data type.

Controlling a mixer group

The best way to control a mixer group is to use the read-modify-write technique. Using this technique, you can examine the group capabilities and ranges before adjusting the group.

The first step in reading the properties and settings of a mixer group is to identify the group. Every mixer group has a name, but because two groups may have the same name, a name alone isn't enough to identify a specific mixer group. In order to make groups unique, mixer groups are identified by the combination of name and index. The index is an integer that represents the instance number of the name. In most cases, the index is 0; in the case of two mixer groups with the same name, the first has an index of 0, and the second has an index of 1.

To read a mixer group, call the *snd_mixer_group_read()* (p. 111) function. The arguments to this function are the mixer handle and the group control structure. The group control structure is of type `snd_mixer_group_t` (p. 113); for details about its members, see the Audio Library chapter.

To read a particular group, you must set its name and index in the *gid* substructure (see `snd_mixer_gid_t` (p. 110)) before making the call. If the call to *snd_mixer_group_read()* succeeds, the function fills in the structure with the group's capabilities and current settings.

Now that you have the group capabilities and current settings, you can modify them before you write them back to the mixer group.

To write the changes to the mixer group, call *snd_mixer_group_write()* (p. 116), passing as arguments the mixer handle and the group control structure.

The best mixer group with respect to your PCM subchannel

In a typical mixer, there are many playback mixer group controls, and possibly several that will control the volume and mute of the stream your application is playing.

For example, consider the Sound Blaster Live playing a wave file. Three playback mixer controls adjust the volume of the playback: Master, PCM, and PCM Subchannel. Although each of these groups can control the volume of our playback, some aren't specific to just our stream, and thus have more side effects.

As an example, consider what happens if you increase your wave file volume by using the Master group. If you do this, any other streams—such a CD playback—are affected as well. So clearly, the best group to use is the PCM subchannel, as it affects only your stream. However, on some cards, a subchannel group might not exist, so you need a better method to find the best group.

The best way to figure out which is the best group for a PCM subchannel is to let the driver (i.e., the driver author) do it. You can obtain the identity of the best mixer group for a PCM subchannel by calling *snd_pcm_channel_setup()* (p. 177) or *snd_pcm_plugin_setup()* (p. 266), as shown below:

```
memset (&setup, 0, sizeof (setup));
memset (&group, 0, sizeof (group));
setup.channel = SND_PCM_CHANNEL_PLAYBACK;
setup.mixer_gid = &group.gid;
if ((rtn = snd_pcm_plugin_setup (pcm_handle, &setup)) < 0)
{
    return -1;
}
```

You must initialize the *setup* structure to zero and then set the *mixer_gid* pointer to a storage location for the group identifier.

One thing to note is that the best group may change, depending on the state of the PCM subchannel. Remember that the PCM subchannels aren't allocated to a client until the parameters of the channel are established. Similarly, the subchannel mixer group isn't available until the subchannel is allocated. Using the example of the Sound Blaster Live, the best mixer group before the subchannel is allocated is the PCM group and, after allocation, the PCM Subchannel group.

Finding all mixer groups

You can get a complete list of mixer groups by calling *snd_mixer_groups()*.

You usually call *snd_mixer_groups()* (p. 118) twice: once to get the total number of mixer groups, then a second time to actually read their IDs. The arguments to the call are the mixer handle and a `snd_mixer_group_t` (p. 113) structure. The structure contains a pointer to where the groups' identifiers are to be stored (an array of `snd_mixer_gid_t` (p. 110) structures), and the size of that array. The call fills in the structure with how many identifiers were stored, and indicates if some couldn't be stored because they would exceed the storage size.

Here's a short example (the *snd_strerror()* (p. 285) prints error messages for the sound functions):

```
while (1)
{
    memset (&groups, 0, sizeof (groups));
    if ((ret = snd_mixer_groups (mixer_handle, &groups) < 0))
    {
       fprintf (stderr, "snd_mixer_groups API call - %s",
                snd_strerror (ret));
    }

    mixer_n_groups = groups.groups_over;
    if (mixer_n_groups > 0)
    {
        groups.groups_size = mixer_n_groups;
        groups.pgroups = (snd_mixer_gid_t *) malloc (
           sizeof (snd_mixer_gid_t) * mixer_n_groups);

        if (groups.pgroups == NULL)
            fprintf (stderr, "Unable to malloc group array - %s",
                     strerror (errno));

        groups.groups_over = 0;
        groups.groups = 0;

        if (snd_mixer_groups (mixer_handle, &groups) < 0)
            fprintf (stderr, "No Mixer Groups ");

        if (groups.groups_over > 0)
        {
            free (groups.pgroups);
            continue;
        }
        else
        {
            printf ("sorting GID table \n");
            snd_mixer_sort_gid_table (groups.pgroups, mixer_n_groups,
                snd_mixer_default_weights);
            break;
        }
    }
}
```

Mixer event notification

By default, all mixer applications are required to keep up-to-date with all mixer changes.

Keeping up-to-date with all mixer changes is done by enqueuing a mixer-change event on all applications other than the application making a change. The driver enqueues these events on all applications that have an open mixer handle, unless the application uses the *snd_mixer_set_filter()* (p. 136) API call to mask out events it's not interested in.

Applications use the *snd_mixer_read()* (p. 128) function to read the enqueued mixer events. The arguments to this functions are the mixer handle and a structure of callback functions to call based on the event type.

You can use the *select()* function (see the QNX Neutrino *C Library Reference*) to determine when to call *snd_mixer_read()*. To get the file descriptor to pass to *select()*, call *snd_mixer_file_descriptor()* (p. 102).

Here's a short example:

```
static void mixer_callback_group (void *private_data,
                                  int cmd,
                                  snd_mixer_gid_t * gid)
{
    switch (cmd)
    {
    case SND_MIXER_READ_GROUP_VALUE:
        printf ("Mixer group %s %d changed value \n",
                gid->name, gid->index);
        break;

    case SND_MIXER_READ_GROUP_ADD:
        break;

    case SND_MIXER_READ_GROUP_REMOVE:
        break;
    }
}

int mixer_update (int fd, void *data, unsigned mode)
{
    snd_mixer_callbacks_t callbacks = { 0, 0, 0, 0 };

    callbacks.group = mixer_callback_group;
    snd_mixer_read (mixer_handle, &callbacks);
    return (Pt_CONTINUE);
}

int main (void)
{
    snd_mixer_t *mixer_handle;
    int ret;

    if ((ret = snd_mixer_open (&mixer_handle, 0, 0) < 0))
        printf ("Unable to open/read mixer - %s",
                snd_strerror (ret));

    PtAppAddFd (NULL,
                snd_mixer_file_descriptor (mixer_handle),
                Pt_FD_READ, mixer_update, NULL);
    ...
}
```

Closing the mixer device

Closing the mixer device frees all the resources associated with the mixer handle and shuts down the connection to the sound mixer interface.

To close the mixer handle, simply call *snd_mixer_close()* (p. 90).

Chapter 4
Optimizing Audio

Here are some tips for reducing audio latency:

- Ensure sample rates and data formats are matched between the `libasound` client and the audio hardware, so that there's no need for Soft SRC or any other data conversion in `libasound`.
- Make sure the `libasound` client's reads and writes are in the exact audio fragments/block size, and disable the `libasound` sub-buffering plugin by calling *snd_pcm_plugin_set_disable()* (p. 256):

  ```
  snd_pcm_plugin_set_disable (pcm_handle,
  PLUGIN_DISABLE_BUFFER_PARTIAL_BLOCKS);
  ```

- In the `libasound` client, configure the audio interface to use a smaller audio fragment/block size (if playing to the software mixer, then the fragment size will be locked to the software mixer's fragment size, which is 4 KB by default).

  > Make sure to look at the fragment size returned via the *snd_pcm_plugin_setup()* (p. 266) call, because the audio interface may not be able to exactly satisfy your fragment size request, depending on various factors such as DMA alignment requirements, potentially required data conversions, and so on.

- In the `libasound` client, set the playback start mode to be `SND_PCM_START_GO`, and then issue the "go" command (by calling *snd_pcm_playback_go()* (p. 233)) after you've written two audio fragments/blocks into the interface. For capture, use `SND_PCM_START_DATA`, which enables capture to the client as soon as one fragment of data is available.
- If you're using the sw_mixer, then you must wait until three audio fragments/blocks are written into the audio interface before issuing the "go" command, or else you will risk an underrun.

Chapter 5
Audio Library

This chapter describes all of the supported QNX Sound Architecture (QSA) API functions, in alphabetical order; undocumented calls aren't supported.

The QSA has similarities to the Advanced Linux Sound Architecture (ALSA), but isn't compatible. Though the function names may be the same, there's no guarantee that QSA and ALSA calls behave the same; some definitely don't.

For an overview of what's in the documentation for a function, see the What's in a Function Description? chapter of the QNX Neutrino *C Library Reference*.

snd_card_get_longname()

Find the long name for a given card number

Synopsis:

```
#include <sys/asoundlib.h>

int snd_card_get_longname ( int card,
                            char *name,
                            size_t size );
```

Arguments:

card

The card number.

name

A buffer in which *snd_card_get_longname()* stores the name.

size

The size of the buffer, in bytes.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_card_get_longname()* function gets the long name associated with the given card number, and stores as much of the name as possible in the buffer pointed to by *name*.

Returns:

Zero, or a negative error code.

Errors:

-EINVAL

The card number is invalid, or *name* is `NULL`.

-EACCES

Search permission is denied on a component of the path prefix, or the device exists and the permissions specified are denied.

-EINTR

The *open()* operation was interrupted by a signal.

-EMFILE

Too many file descriptors are currently in use by this process.

-ENFILE

Too many files are currently open in the system.

-ENOENT

The named device doesn't exist.

-ENOMEM

No memory available for data structure.

-SND_ERROR_INCOMPATIBLE_VERSION

The audio driver version is incompatible with the client library that the application is using.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_card_get_name()

Find the name for a given card number

Synopsis:

```
#include <sys/asoundlib.h>

int snd_card_get_name( int card,
                       char *name,
                       size_t size );
```

Arguments:

card

The card number.

name

A buffer in which *snd_card_get_name()* stores the name.

size

The size of the buffer, in bytes.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_card_get_name()* function gets the common name that's associated with the given card number, and stores as much of the name as possible in the buffer pointed to by *name*.

Returns:

Zero, or a negative error code.

Errors:

-EINVAL

The card number is invalid, or *name* is `NULL`.

-EACCES

Search permission is denied on a component of the path prefix, or the device exists and the permissions specified are denied.

-EINTR

The *open()* operation was interrupted by a signal.

-EMFILE

Too many file descriptors are currently in use by this process.

-ENFILE

Too many files are currently open in the system.

-ENOENT

The named device doesn't exist.

-ENOMEM

No memory available for data structure.

-SND_ERROR_INCOMPATIBLE_VERSION

The audio driver version is incompatible with the client library that the application is using.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_card_name()

Find the card number for a given name

Synopsis:

```
#include <sys/asoundlib.h>

int snd_card_name ( const char *string );
```

Arguments:

string

The name of the card.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_card_name()* function returns the card number associated with the given card name.

Returns:

A card number (positive integer), or a negative error code.

Errors:

`-EINVAL`

The *string* argument is `NULL`, an empty string, or isn't the name of a card.

`-EACCES`

Search permission is denied on a component of the path prefix, or the device exists and the permissions specified are denied.

`-EINTR`

The *open()* operation was interrupted by a signal.

`-EMFILE`

Too many file descriptors are currently in use by this process.

-ENFILE

Too many files are currently open in the system.

-ENOENT

The named device doesn't exist.

-ENOMEM

No memory available for data structure.

-SND_ERROR_INCOMPATIBLE_VERSION

The audio driver version is incompatible with the client library that the application is using.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_cards()

Count the sound cards

Synopsis:

```
#include <sys/asoundlib.h>

int snd_cards ( void );
```

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_cards()* function returns the instantaneous number of sound cards that have running drivers. There's no guarantee that the sound cards have contiguous card numbers, and cards may be unmounted at any time.

This function is mainly provided for historical reasons. You should use *snd_cards_list()* (p. 61) instead.

Returns:

The number of sound cards.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_cards_list()

Count the sound cards and list their card numbers in an array

Synopsis:

```
#include <sys/asoundlib.h>

int snd_cards_list( int *cards,
                    int card_array_size,
                    int *cards_over );
```

Arguments:

cards

An array in which *snd_cards_list()* stores the card numbers.

card_array_size

The number of card numbers that the array *cards* can hold.

cards_over

The number of cards that wouldn't fit in the *cards* array.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_cards_list()* function returns the instantaneous number of sound cards that have running drivers. There's no guarantee that the sound cards have contiguous card numbers, and cards may be unmounted at any time.

You should use this function instead of *snd_cards()* (p. 60) because *snd_cards_list()* can fill in an array of card numbers. This overcomes the difficulties involved in hunting a (possibly) non-contiguous list of card numbers for active cards.

Returns:

The number of sound cards.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_ctl_callbacks_t

Control callback functions

Synopsis:

```
typedef struct snd_ctl_callbacks {
    void *private_data; /* should be used by an application */
    void (*rebuild) (void *private_data);
    void (*xswitch) (void *private_data, int cmd,
                     int iface, snd_switch_list_item_t *item);
    void *reserved[29]; /* reserved for future use - must be NULL!!! */
} snd_ctl_callbacks_t;
```

Description:

Use the `snd_ctl_callbacks_t` structure to define the callback functions that you need to handle control events. Pass a pointer to an instance of this structure to *snd_ctl_read()* (p. 85).

The members of the `snd_ctl_callbacks_t` structure include:

- *private_data*, a pointer to arbitrary data that you want to pass to the callbacks
- pointers to the callbacks, which are described below.

Make sure that you zero-fill any members that you aren't interested in. You can zero-fill the entire `snd_ctl_callbacks_t` structure if you aren't interested in tracking any of these events.

***rebuild* callback**

The *rebuild* callback is called whenever the control device is rebuilt. Its only argument is the *private_data* that you specified in this structure.

***xswitch* callback**

The *xswitch* callback is called whenever a switch changes. Its arguments are:

private_data

A pointer to the arbitrary data that you specified in this structure.

cmd

One of:

- `SND_CTL_READ_SWITCH_VALUE`
- `SND_CTL_READ_SWITCH_CHANGE`
- `SND_CTL_READ_SWITCH_ADD`
- `SND_CTL_READ_SWITCH_REMOVE`

iface

The device interface the switch is natively associated with. The possible values are (from `<sys/asound.h>`):

- `SND_CTL_IFACE_CONTROL`
- `SND_CTL_IFACE_MIXER`
- `SND_CTL_IFACE_PCM_PLAYBACK`
- `SND_CTL_IFACE_PCM_CAPTURE`

item

A pointer to a `snd_switch_list_item_t` structure that identifies the specific switch that's been changed. This structure has only a *name* member.

Classification:

QNX Neutrino

snd_ctl_close()

Close a control handle

Synopsis:

```
#include <sys/asoundlib.h>

int snd_ctl_close( snd_ctl_t *handle );
```

Arguments:

handle

The handle for the control connection to the card. This must be a handle created by *snd_ctl_open()* (p. 79).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_close()* function frees all the resources allocated with the control handle and closes the connection to the control interface.

Returns:

Zero on success, or a negative value on error.

Errors:

-EBADF

Invalid file descriptor. Your *handle* may be corrupt.

-EINTR

The *close()* call was interrupted by a signal.

-EINVAL

Invalid *handle* argument.

-EIO

An I/O error occurred while updating the directory information.

-ENOSPC

A previous buffered write call has failed.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_ctl_file_descriptor()

Get the control file descriptor

Synopsis:

```
#include <sys/asoundlib.h>

int snd_ctl_file_descriptor( snd_ctl_t *handle );
```

Arguments:

handle

The handle for the control connection to the card. This must be a handle created by *snd_ctl_open()* (p. 79).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_file_descriptor()* function returns the file descriptor of the connection to the control interface.

You can use the file descriptor for the *select()* function (see the QNX Neutrino *C Library Reference*) for determining if something can be read or written. Your application should then call *snd_ctl_read()* (p. 85) if data is waiting to be read.

Returns:

The file descriptor of the connection to the control interface, or a negative value if an error occurs.

Errors:

`-EINVAL`

Invalid *handle* argument.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_ctl_hw_info()

Get information about a sound card's hardware

Synopsis:

```
#include <sys/asoundlib.h>

int snd_ctl_hw_info( snd_ctl_t *handle,
                     struct snd_ctl_hw_info *info );
```

Arguments:

handle

The handle for the control connection to the card. This must be a handle created by *snd_ctl_open()* (p. 79).

info

A pointer to a `snd_ctl_hw_info_t` (p. 71) structure in which *snd_ctl_hw_info()* stores the information.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_hw_info()* function fills the *info* structure with information about the sound card hardware selected by handle.

Returns:

Zero on success, or a negative value if an error occurs.

Errors:

-EBADF

Invalid file descriptor. Your *handle* may be corrupt.

-EINVAL

Invalid *handle* argument.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_ctl_hw_info_t

Information about a sound card's hardware

Synopsis:

```
typedef struct snd_ctl_hw_info
{
    uint32_t    type;
    uint32_t    hwdepdevs;
    uint32_t    pcmdevs;
    uint32_t    mixerdevs;
    uint32_t    mididevs;
    uint32_t    timerdevs;
    char        id[16];
    char        abbreviation[16];
    char        name[32];
    char        longname[80];
    uint8_t     reserved[128];      /* must be filled with
zeroes */
}       snd_ctl_hw_info_t;
```

Description:

The `snd_ctl_hw_info_t` structure describes a sound card's hardware. You can get this information by calling *snd_ctl_hw_info()* (p. 69).

The members include:

type

The type of sound card. Deprecated; don't use this member.

hwdepdevs

The total number of hardware-dependent devices on this sound card. Deprecated; don't use this member.

pcmdevs

The total number of PCM devices on this sound card.

mixerdevs

The total number of mixer devices on this sound card.

mididevs

The total number of midi devices on this sound card. Not supported at this time; don't use this member.

timerdevs

The total number of timer devices on this sound card. Not supported at this time; don't use this member.

id

An ID string that identifies this sound card.

abbreviation

An abbreviated name for identifying this sound card.

name

A common name for this sound card.

longname

A unique, descriptive name for this sound card.

reserved

Reserved; this member must be filled with zeroes.

Classification:

QNX Neutrino

snd_ctl_mixer_switch_list()

Get the number and names of control switches for the mixer

Synopsis:

```
#include <sys/asoundlib.h >
int snd_ctl_mixer_switch_list( snd_ctl_t *handle,
                              int dev, snd_switch_list_t *list );
```

Arguments:

handle

The handle for the control device. This must have been created by *snd_ctl_open()* (p. 79).

dev

The mixer device the switches apply to.

list

A pointer to a `snd_switch_list_t` structure that *snd_ctl_mixer_switch_list()* fills with information about the switch.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_mixer_switch_list()* function uses the control device handle to fill the given `snd_switch_list_t` structure with the number of switches for the mixer specified. It also fills in the array of switches pointed to by *pswitches* to a limit of *switches_size*. Before calling *snd_mixer_groups()*, set the members of the `snd_switch_list_t` as follows:

pswitches

This pointer must be `NULL` or point to a valid storage location for the switches (i.e., an array of `snd_switch_list_item_t` structures).

switches_size

The size of the *pswitches* storage location in `sizeof( snd_switch_list_item_t )` units (i.e., the number of entries in the array).

On a successful return, the *snd_ctl_mixer_switch_list()* function will fill in these members:

switches

The total switches in this mixer device.

switches_over

The number of switches that couldn't be copied to the storage location.

Returns:

Zero on success, or a negative value if an error occurs.

Errors:

`-EINVAL`

Invalid *handle* argument.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The switch struct must be initialized to a known state before making the call; use *memset()* to set the struct to zero, and then set the name member to specify which switch to read.

snd_ctl_mixer_switch_read()

Get a mixer switch setting

Synopsis:

```
#include <sys/asoundlib.h >

int snd_ctl_mixer_switch_read(
       snd_ctl_t *handle,
        int dev,
        snd_switch_t * sw )
```

Arguments:

handle

The handle for the control device. This must have been created by *snd_ctl_open()* (p. 79).

dev

The mixer device the switches apply to.

sw

A pointer to a `snd_switch_t` structure that *snd_ctl_mixer_switch_read()* fills with information about the switch.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_mixer_switch_read()* function reads the `snd_switch_t` structure for the switch identified by the *name* member of the structure.

> You must initialize the name member before calling this function.

Returns:

Zero on success, or a negative value if an error occurs.

Errors:

-EINVAL

Invalid *handle* argument.

-ENXIO

The group wasn't found.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_ctl_mixer_switch_write()

Adjust a mixer switch setting

Synopsis:

```
#include  < sys/asoundlib.h >

int snd_ctl_mixer_switch_write(
       snd_ctl_t *handle,
        int dev,
        snd_switch_t * sw )
```

Arguments:

handle

The handle for the control device. This must have been created by *snd_ctl_open()* (p. 79).

dev

The mixer device the switches apply to.

sw

A pointer to a `snd_switch_t` structure that *snd_ctl_mixer_switch_write()* writes to the driver about the switch.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_mixer_switch_write()* function writes the `snd_switch_t` structure for the switch identified by the structure's *name* member.

Returns:

Zero on success, or a negative value if an error occurs.

Errors:

`-EINVAL`

Invalid *handle* argument.

-ENXIO

The group wasn't found.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The switch struct must be initialized completely before making the call.

snd_ctl_open()

Create a connection and handle to the specified control device

Synopsis:

```
#include <sys/asoundlib.h>

int snd_ctl_open( snd_ctl_t **handle,
                  int card );
```

Arguments:

handle

A pointer to a location in which *snd_ctl_open()* stores a handle for the card, which you need to pass to the other *snd_ctl_** functions.

card

The card number.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_open()* function creates a new handle and opens a connection to the control interface for sound card number *card* (0-N). This handle may be used in all of the other *snd_ctl_*()* calls.

Returns:

Zero on success, or a negative value if an error occurs.

Errors:

-EACCES

Search permission is denied on a component of the path prefix, or the device exists and the permissions specified are denied.

-EINTR

The *open()* operation was interrupted by a signal.

-EMFILE

Too many file descriptors are currently in use by this process.

-ENFILE

Too many files are currently open in the system.

-ENOENT

The named device doesn't exist.

-ENOMEM

No memory available for data structure.

-SND_ERROR_INCOMPATIBLE_VERSION

The audio driver version is incompatible with the client library that the application is using.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_ctl_pcm_channel_info()

Get information about a PCM channel's capabilities from a control handle

Synopsis:

```
#include <sys/asoundlib.h>

int snd_ctl_pcm_channel_info(
        snd_ctl_t *handle,
        int dev,
        int chn,
        int subdev,
        snd_pcm_channel_info_t *info );
```

Arguments:

handle

The handle for the control connection to the card. This must be a handle created by *snd_ctl_open()* (p. 79).

dev

The PCM device number.

chn

The channel direction; either `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

subdev

The PCM subchannel.

info

A pointer to a `snd_pcm_channel_info_t` (p. 161) structure in which *snd_ctl_pcm_channel_info()* stores the information.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_pcm_channel_info()* function fills the *info* structure with data about the PCM subchannel *subdev* in the PCM channel *chn* on the sound card selected by *handle*.

This function gets information about the complete capabilities of the system. It's similar to *snd_pcm_channel_info()* (p. 159) and *snd_pcm_plugin_info()* (p. 245), but these functions get a dynamic "snapshot" of the system's current capabilities, which can shrink and grow as subchannels are allocated and freed.

Returns:

Zero on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_ctl_pcm_info()

Get general information about a PCM device from a control handle

Synopsis:

```
#include <sys/asoundlib.h>

int snd_ctl_pcm_info( snd_ctl_t *handle,
                      int dev,
                      snd_pcm_info_t *info );
```

Arguments:

handle

The handle for the control connection to the card. This must be a handle created by *snd_ctl_open()* (p. 79).

dev

The PCM device.

info

A pointer to a `snd_pcm_info_t` (p. 216) structure in which *snd_ctl_pcm_info()* stores the information.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_pcm_info()* function fills the *info* structure with information about the capabilities of the PCM device *dev* on the sound card selected by *handle*.

Returns:

Zero on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_ctl_read()

Read pending control events

Synopsis:

```
#include <sys/asoundlib.h>

int snd_ctl_read( snd_ctl_t *handle,
                  snd_ctl_callbacks_t *callbacks );
```

Arguments:

handle

The handle for the control connection to the card. This must be a handle created by *snd_ctl_open()* (p. 79).

callbacks

A pointer to a `snd_ctl_callbacks_t` (p. 63) structure that defines the callbacks for the events.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_ctl_read()* function reads pending control events from the control handle. As each event is read, the list of callbacks is checked for a handler for this event. If a match is found, the callback is invoked. This function is usually called on the return of the *select()* library call (see the QNX Neutrino *C Library Reference*).

If you register to receive notification of events (e.g., by using *select()*), it's very important that you clear the event queue by calling *snd_ctl_read()*, even if you don't want or need the information. The event queues are open-ended and may cause trouble if allowed to grow in an uncontrolled manner. The best practice is to read the events in the queues as you receive notification, so that they don't have a chance to accumulate.

Returns:

The number of events read from the handle, or a negative value on error.

Errors:

-EBADF

Invalid file descriptor. Your *handle* may be corrupt.

-EINTR

The read operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

-EIO

An event I/O error occurred.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_callbacks_t

List of mixer callback functions

Synopsis:

```
typedef struct snd_mixer_callbacks {
    void *private_data; /* should be used with an application */
    void (*rebuild) (void *private_data);
    void (*element) (void *private_data, int cmd,
                     snd_mixer_eid_t *eid);
    void (*group) (void *private_data, int cmd,
                   snd_mixer_gid_t *gid);
    void *reserved[28]; /* reserved for future use - must be NULL!!! */
} snd_mixer_callbacks_t;
```

Description:

The `snd_mixer_callbacks_t` structure defines a list of callbacks that you can provide to handle events read by *snd_mixer_read()* (p. 128). The members include:

- *private_data*, a pointer to arbitrary data that you want to pass to the callbacks
- pointers to the callbacks, which are described below.

Make sure that you zero-fill any members that you aren't interested in. You can zero-fill the entire `snd_mixer_callbacks_t` structure if you aren't interested in tracking any of these events. The `wave.c` *example* does this.

---

rebuild callback

The *rebuild* callback is called whenever the mixer is rebuilt. Its only argument is the *private_data* that you specified in this structure.

element callback

The *element* callback is called whenever an element event occurs. The arguments to this function are:

private_data

A pointer to the arbitrary data that you specified in this structure.

cmd

A `SND_MIXER_READ_ELEMENT_*` event code:

- `SND_MIXER_READ_ELEMENT_VALUE` — the element's value changed.
- `SND_MIXER_READ_ELEMENT_CHANGE` — the element changed (something other than its value).

- SND_MIXER_READ_ELEMENT_ADD — the element was added (i.e., created).
- SND_MIXER_READ_ELEMENT_REMOVE — the element was removed (i.e., destroyed).
- SND_MIXER_READ_ELEMENT_ROUTE — the element's routing information changed.

eid

A pointer to a snd_mixer_eid_t (p. 92) structure that holds the ID of the element affected by the event.

group callback

The *group* callback is called whenever a group event occurs. The arguments are:

private_data

A pointer to the arbitrary data that you specified in this structure.

cmd

A SND_MIXER_READ_GROUP_* event code:

- SND_MIXER_READ_GROUP_VALUE — the group's value changed.
- SND_MIXER_READ_GROUP_CHANGE — the group changed (something other than the value).
- SND_MIXER_READ_GROUP_ADD — the group was added (i.e., created).
- SND_MIXER_READ_GROUP_REMOVE — the group was removed (i.e., destroyed).

gid

A pointer to a snd_mixer_gid_t (p. 110) structure that holds the ID of the group affected by the event.

Examples:

```
static void
mixer_callback_group (void *private_data, int cmd, snd_mixer_gid_t * gid)
{
    Control_t *control, *prev;
    PtWidget_t *above_wgt;
    int     i;

    switch (cmd)
    {
    case SND_MIXER_READ_GROUP_VALUE:
        for (control = control_head; control; control = control->next)
        {
            if (strcmp (control->group.gid.name, gid->name) == 0 &&
                control->group.gid.index == gid->index)
            {
```

```
                if (snd_mixer_group_read (mixer_handle, &control->group) == 0)
                    base_update_control (control, NULL);
            }
        }
        break;

    case SND_MIXER_READ_GROUP_ADD:
        if ((control = mixer_create_control (gid, control_tail)))
        {
            if (control->group.caps & SND_MIXER_GRPCAP_PLAY_GRP)
                above_wgt = PtWidgetBrotherBehind (ABW_base_capture_pane);
            else
                above_wgt = PtWidgetBrotherBehind (ABW_base_status);
            PtContainerHold (ABW_base_controls);
            base_create_control (ABW_base_controls, &above_wgt, control);
            PtContainerRelease (ABW_base_controls);
        }
        break;

    case SND_MIXER_READ_GROUP_REMOVE:
        for (prev = NULL, control = control_head; control;
                            prev = control, control = control->next)
        {
            if (strcmp (control->group.gid.name, gid->name) == 0 &&
                control->group.gid.index == gid->index)
                mixer_delete_control (control, prev);
        }
        break;
    }

}


int
mixer_update (int fd, void *data, unsigned mode)
{
    snd_mixer_callbacks_t callbacks = { 0, 0, 0, 0 };

    callbacks.group = mixer_callback_group;
    snd_mixer_read (mixer_handle, &callbacks);
    return (Pt_CONTINUE);
}
```

Classification:

QNX Neutrino

snd_mixer_close()

Close a mixer handle

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_close( snd_mixer_t *handle );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_close()* function frees all the resources allocated with the mixer handle and closes the connection to the sound mixer interface.

Returns:

Zero, or a negative value on error.

Errors:

-EINTR

The *close()* call was interrupted by a signal.

-EINVAL

Invalid *handle* argument.

-EIO

An I/O error occurred while updating the directory information.

-ENOSPC

A previous buffered write call has failed.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_eid_t

Mixer element ID structure

Synopsis:

```
ypedef struct
{
    int32_t     type;
    char        name[36];
    int32_t     index;
    uint8_t     reserved[120];      /* must be filled with
zeroes */
    int32_t     weight;
}       snd_mixer_eid_t;
```

Description:

The `snd_mixer_eid_t` structure describes a mixer element's ID. The members include:

type

The type of element.

name

The name of the element.

index

The index of the element.

weight

Reserved for internal sorting operations.

We recommend that you work with mixer groups instead of manipulating the elements directly.

Classification:

QNX Neutrino

snd_mixer_element_read()

Get a mixer element's configurable parameters

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_element_read(
        snd_mixer_t *handle,
        snd_mixer_element_t *element );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

element

A pointer to a `snd_mixer_element_t` (p. 95) in which *snd_mixer_element_read()* stores the element's configurable parameters.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_element_read()* function fills the `snd_mixer_element_t` structure with information on the current settings of the element identified by the `eid` substructure.

We recommend that you work with mixer groups instead of manipulating the elements directly.

Returns:

Zero on success, or a negative error value on error.

Errors:

`-EINVAL`

Invalid *handle* or *element* argument.

-ENXIO

The element wasn't found.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The `element` struct must be initialized to a known state before making the call: use *memset()* to set the struct to zero, and then set the `eid` member to specify which element to read.

snd_mixer_element_t

Mixer element control structure

Synopsis:

```
typedef struct snd_mixer_element
{
    snd_mixer_eid_t eid;
    union
    {
        snd_mixer_element_switch1           switch1;
        snd_mixer_element_switch2           switch2;
        snd_mixer_element_switch3           switch3;
        snd_mixer_element_volume1           volume1;
        snd_mixer_element_volume2           volume2;
        snd_mixer_element_accu3             accu3;
        snd_mixer_element_mux1              mux1;
        snd_mixer_element_mux2              mux2;
        snd_mixer_element_tone_control1     tc1;
        snd_mixer_element_3d_effect1        teffect1;
        snd_mixer_element_pan_control1      pc1;
        snd_mixer_element_pre_effect1       peffect1;
        uint8_t                             reserved[128];
            /* must be filled with zeroes */
    }       data;
    uint8_t     reserved[128];      /* must be filled with
zeroes */
}       snd_mixer_element_t;
```

Description:

The `snd_mixer_element_t` structure contains the settings associated with a mixer element.

We recommend that you work with mixer groups instead of manipulating the elements directly.

Classification:

QNX Neutrino

snd_mixer_element_write()

Set a mixer element's configurable parameters

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_element_write(
        snd_mixer_t *handle,
        snd_mixer_element_t *element );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

element

A pointer to a `snd_mixer_element_t` (p. 95) from which *snd_mixer_element_read()* sets the element's configurable parameters.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_element_write()* function writes the given `snd_mixer_element_t` structure to the driver.

We recommend that you work with mixer groups instead of manipulating the elements directly.

Returns:

Zero on success, or a negative value on error.

Errors:

-EBUSY

The element has been modified by another application.

-EINVAL

Invalid *handle* or *element* argument.

-ENXIO

The element wasn't found.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The write may fail with -EBUSY if another application has modified the element, and this application hasn't read that event yet using *snd_mixer_read()* (p. 128).

snd_mixer_elements()

Get the number of elements in the mixer and their element IDs

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_elements(
        snd_mixer_t *handle,
        snd_mixer_elements_t *elements );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

elements

A pointer to a `snd_mixer_elements_t` (p. 100) structure in which *snd_mixer_elements()* stores the information about the elements.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_elements()* function fills the given `snd_mixer_elements_t` structure with the number of elements in the mixer that the handle was opened on. It also fills in the array of element IDs pointed to by *pelements* to a limit of *elements_size*.

We recommend that you work with mixer groups instead of manipulating the elements directly.

Before calling *snd_mixer_elements()*, set the `snd_mixer_elements_t` structure as follows:

pelements

This pointer be `NULL`, or point to a valid storage location for the elements (i.e., an array of `snd_mixer_eid_t` (p. 92) structures).

elements_size

This must reflect the size of the *pelements* storage location, in `sizeof( snd_mixer_eid_t )` units (i.e., *elements_size* must be the number of entries in the *pelements* array).

On a successful return, *snd_mixer_elements()* sets these members:

elements

The total number of elements in the mixer.

pelements

If non-`NULL`, the mixer element IDs are filled in.

elements_over

The number of elements that couldn't be copied to the storage location.

Returns:

Zero on success, or a negative value on error.

Errors:

`-EINVAL`

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_elements_t

Information about all elements in a mixer

Synopsis:

```
typedef struct snd_mixer_elements_s
{
    int32_t     elements, elements_size, elements_over;
    uint8_t     zero[4];             /* alignment -- zero fill
 */
    snd_mixer_eid_t *pelements;
    void        *pzero;              /* align pointers on
64-bits;
                                        point to NULL */
    uint8_t     reserved[128];       /* must be filled with
zeroes */
}       snd_mixer_elements_t;
```

Description:

The `snd_mixer_elements_t` structure describes all the elements in a mixer. You can fill in this structure by calling *snd_mixer_elements()* (p. 98).

We recommend that you work with mixer groups instead of manipulating the elements directly.

The members of the `snd_mixer_elements_t` structure include:

elements

The total number of elements in the mixer.

elements_size

The size of the *pelements* storage location, in `sizeof(snd_mixer_eid_t)` units (i.e., the number of entries in the *pelements* array). Set this element before calling *snd_mixer_elements()*.

elements_over

The number of elements that couldn't be copied to the storage location.

pelements

`NULL`, or a pointer to an array of `snd_mixer_eid_t` (p. 92) structures.

If *pelements* isn't `NULL`, *snd_mixer_elements()* stores the mixer element IDs in the array.

Classification:

QNX Neutrino

snd_mixer_file_descriptor()

Return the file descriptor of the connection to the sound mixer interface

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_file_descriptor(
        snd_mixer_t *handle );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_file_descriptor()* function returns the file descriptor of the connection to the sound mixer interface.

You should use this file descriptor with the *select()* synchronous multiplexer function (see the QNX Neutrino *C Library Reference*) to receive notification of mixer events. If data is waiting to be read, you can read in the events with *snd_mixer_read()* (p. 128).

Returns:

The file descriptor of the connection to the mixer interface on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle* argument.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_filter_t

Information about a mixer's filters

Synopsis:

```
typedef struct snd_mixer_filter
{
    uint32_t    enable;          /* bitfield of 1 << SND_MIXER_READ_* */
    uint8_t     reserved[124];   /* must be filled with zeroes */
}       snd_mixer_filter_t;
```

Description:

The `snd_mixer_filter_t` structure describes the filters for a mixer. You can call *snd_mixer_set_filter()* (p. 136) to specify the events you want to track, and *snd_mixer_get_filter()* (p. 108) to determine which you're tracking.

Currently, the only member of this structure is *enable*, which is a mask of the mixer events. The bits in the mask include:

SND_MIXER_READ_REBUILD

The mixer has been rebuilt.

SND_MIXER_READ_ELEMENT_VALUE

An element's value has changed.

SND_MIXER_READ_ELEMENT_CHANGE

An element has changed in some way other than its value.

SND_MIXER_READ_ELEMENT_ADD

An element was added to the mixer.

SND_MIXER_READ_ELEMENT_REMOVE

An element was removed from the mixer.

SND_MIXER_READ_ELEMENT_ROUTE

A route was added or changed.

SND_MIXER_READ_GROUP_VALUE

A group's value has changed.

SND_MIXER_READ_GROUP_CHANGE

A group has changed in some way other than its value.

SND_MIXER_READ_GROUP_ADD

A group was added to the mixer.

SND_MIXER_READ_GROUP_REMOVE

A group was removed from the mixer.

Classification:

QNX Neutrino

snd_mixer_get_bit()

Return the boolean value of a single bit in the specified bitmap

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_get_bit( unsigned int *bitmap,
                       int bit );
```

Arguments:

bitmap

The bitmap to test. Note that *bitmap* is an array and may be longer than 32 bits.

bit

The index into *bitmap* of the bit to get.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_get_bit()* function is a convenience function that returns the value (0 or 1) of the bit specified by *bit* in the *bitmap*.

Returns:

The value of the specified bit (0 or 1).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

snd_mixer_get_filter()

Get the current mask of mixer events that the driver is tracking

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_get_filter(
       snd_mixer_t *handle,
       snd_mixer_filter_t *filter );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

filter

A pointer to a `snd_mixer_filter_t` (p. 104) structure that *snd_mixer_get_filter()* fills in with the mask.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_get_filter()* function fills the `snd_mixer_filter_t` structure with a mask of all mixer events for the mixer that the handle was opened on that the driver is tracking.

You can arrange to have your application receive notification when an event occurs by calling *select()* on the mixer's file descriptor, which you can get by calling *snd_mixer_file_descriptor()* (p. 102). You can use *snd_mixer_read()* (p. 128) to read the event's data.

Returns:

Zero on success, or a negative value on error.

Errors:

`-EINVAL`

Invalid *handle* or *filter* is NULL.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_gid_t

Mixer group ID structure

Synopsis:

```
typedef struct
{
    int32_t     type;
    char        name[32];
    int32_t     index;
    uint8_t     reserved[124];      /* must be filled with
zeroes */
    int32_t     weight;
}       snd_mixer_gid_t;
```

Description:

The `snd_mixer_gid_t` structure describes a mixer group's ID. The members include:

type

The group's type. Not currently used; set it to 0.

name

The group's name.

index

The group's index number.

weight

Reserved for internal sorting operations.

Classification:

QNX Neutrino

snd_mixer_group_read()

Get a mixer group's configurable parameters

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_group_read(
        snd_mixer_t *handle,
        snd_mixer_group_t *group );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

group

A pointer to a `snd_mixer_group_t` (p. 113) structure that *snd_mixer_group_read()* fills in with information about the mixer group.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_group_read()* function reads the `snd_mixer_group_t` (p. 113) structure for the group identified by the *gid* substructure (for more information, see `snd_mixer_gid_t` (p. 110)).

You must initialize the *gid* substructure before calling this function.

Returns:

Zero on success, or a negative error value on error.

Errors:

`-EINVAL`

Invalid *handle* argument.

-ENXIO

The group wasn't found.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The `group` struct must be initialized to a known state before making the call: use *memset()* to set the struct to zero, and then set the `gid` member to specify which group to read.

snd_mixer_group_t

Mixer group control structure

Synopsis:

```
typedef struct snd_mixer_group_s
{
    snd_mixer_gid_t gid;
    uint32_t    caps;
    uint32_t    channels;
    int32_t     min, max;
    union
    {
        uint32_t    values[32];
        struct
        {
            uint32_t    front_left;
            uint32_t    front_right;
            uint32_t    front_center;
            uint32_t    rear_left;
            uint32_t    rear_right;
            uint32_t    woofer;
            uint8_t     reserved[128]; /* must be filled with zeroes */
        }       names;
    }       volume;
    uint32_t    mute;
    uint32_t    capture;
    int32_t     capture_group;

    int32_t     elements_size, elements, elements_over;
    snd_mixer_eid_t *pelements;
    uint16_t    change_duration;    /* milliseconds */
    uint16_t    spare;
    int32_t     min_dB, max_dB;
    uint8_t     reserved[120];      /* must be filled with zeroes */
}       snd_mixer_group_t;
```

Description:

The `snd_mixer_group_t` structure is the control structure for a mixer group. You can get the information for a group by calling *snd_mixer_group_read()* (p. 111), and set it by calling *snd_mixer_group_write()* (p. 116).

The members of this structure include:

gid

A `snd_mixer_gid_t` (p. 110) structure that identifies the group. This structure includes the group name and index.

caps

The capabilities of the group, expressed through any combination of these flags:

- `SND_MIXER_GRPCAP_VOLUME` — the group has at least one volume control.

- `SND_MIXER_GRPCAP_JOINTLY_VOLUME` — all channel volume levels for the group must be the same (ganged).
- `SND_MIXER_GRPCAP_MUTE` — the group has at least one mute control.
- `SND_MIXER_GRPCAP_JOINTLY_MUTE` — all channel mute settings for the group must be the same (ganged).
- `SND_MIXER_GRPCAP_CAPTURE` — the group can be captured (recorded).
- `SND_MIXER_GRPCAP_JOINTLY_CAPTURE` — all channel capture settings for the group must be the same (ganged).
- `SND_MIXER_GRPCAP_EXCL_CAPTURE` — only one group on this device can be captured at a time.
- `SND_MIXER_GRPCAP_PLAY_GRP` — the group is a playback group.
- `SND_MIXER_GRPCAP_CAP_GRP` — the group is a capture group.
- `SND_MIXER_GRPCAP_SUBCHANNEL` — the group is a subchannel control. It exists only while a PCM subchannel is allocated by an application.

channels

The mapped bits that correspond to the channels contained in this group.

For example, for stereo right and left speakers, bits 1 and 2 (00011) are mapped; for the center speaker, bit 3 (00100) is mapped.

min, max

The minimum and maximum values that define the volume range. Note that the minimum doesn't have to be zero.

volume

A structure that contains the volume level for each channel in the group. You can access the values accessed directly by name or indirectly through the array of values.

If the group is jointly volumed, all volume values must be the same; setting different values results in undefined behavior.

mute

The mute state of the group channels. If the bit corresponding to the channel is set, the channel is muted.

If the group is jointly muted, all mute bits must be the same; setting the bits differently results in undefined behavior.

capture

The capture state of the group channels. If the bit corresponding to the channel is set, the channel is being captured. If the group is exclusively capture, setting capture on this group means that another group is no longer being captured.

If the group is jointly captured, all capture bits must be the same; setting the bits differently results in undefined behavior.

capture_group

Not currently used.

elements_size

The size of the memory block pointed to by *pelements* in units of `snd_mixer_eid_t`.

elements

The number of element IDs that are currently valid in *pelements*.

elements_over

The number of element IDs that were not returned in *pelements* because it wasn't large enough.

pelements

A pointer to a region of memory (allocated by the calling application) that's used to store an array of element IDs. This is an array of `snd_mixer_eid_t` (p. 92) structures.

The elements that are returned are the component elements that make up the group identified by *gid*.

change_duration

The number of milliseconds over which to ramp the volume.

min_dB*, *max_dB

The minimum and maximum sound levels, in decibels.

Classification:

QNX Neutrino

snd_mixer_group_write()

Set a mixer group's configurable parameters

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_group_write(
        snd_mixer_t *handle,
        snd_mixer_group_t *group );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

group

A pointer to a `snd_mixer_group_t` (p. 113) structure that contains the information you want to set for the mixer group.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_group_write()* function writes the `snd_mixer_group_t` structure to the driver. This structure contains the volume levels and mutes associated with the group.

Returns:

Zero on success, or a negative value on error.

Errors:

-EBUSY

The group has been modified by another application.

-EINVAL

Invalid *handle* argument.

-ENXIO

The group wasn't found.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

The write may fail with -EBUSY if another application has modified the group, and this application hasn't read that event yet using *snd_mixer_read()* (p. 128).

snd_mixer_groups()

Get the number of groups in the mixer and their group IDs

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_groups( snd_mixer_t *handle,
                      snd_mixer_groups_t *groups );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

groups

A pointer to a `snd_mixer_groups_t` (p. 120) structure that *snd_mixer_groups()* fills in with information about the groups.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_groups()* function fills the given `snd_mixer_groups_t` structure with the number of groups in the mixer that the handle was opened on. It also fills in the array of group IDs pointed to by *pgroups* to a limit of *groups_size*.

Before calling *snd_mixer_groups()*, set the members of the `snd_mixer_groups_t` as follows:

pgroups

This pointer must be `NULL` or point to a valid storage location for the groups (i.e., an array of `snd_mixer_gid_t` (p. 110) structures).

groups_size

The size of the *pgroups* storage location in `sizeof(snd_mixer_gid_t)` units (i.e., the number of entries in the array).

On a successful return, *snd_mixer_groups()* fills in these members:

groups

The total groups in the mixer.

groups_over

The number of groups that couldn't be copied to the storage location.

Returns:

Zero on success, or a negative value on error.

Errors:

`-EINVAL`

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_groups_t

Information about all of the mixer groups

Synopsis:

```
typedef struct snd_mixer_groups_s
{
    int32_t     groups, groups_size, groups_over;
    uint8_t     zero[4];        /* alignment -- zero fill */
    snd_mixer_gid_t *pgroups;
    void        *pzero;         /* align pointers on 64-bits;
                                   point to NULL */
    uint8_t     reserved[128];  /* must be filled with zeroes
 */
}       snd_mixer_groups_t;
```

Description:

The `snd_mixer_groups_t` structure holds information about all of the mixer groups. You can fill this structure by calling *snd_mixer_groups()* (p. 118).

The members of this structure include:

groups

The number of groups in the mixer.

groups_size

The size of the *pgroups* storage location in `sizeof(snd_mixer_gid_t)` units (i.e., the number of entries in the array). Set this before calling *snd_mixer_groups()*.

groups_over

The number of groups that wouldn't fit in the *pgroups* array.

pgroups

`NULL`, or an array of `snd_mixer_gid_t` (p. 110) structures.

If *pgroups* isn't `NULL`, *snd_mixer_groups()* stores the group IDs in the array.

Classification:

QNX Neutrino

snd_mixer_info()

Get general information about a mixer device

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_info( snd_mixer_t *handle,
                    snd_mixer_info_t *info );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

info

A pointer to a `snd_mixer_info_t` (p. 123) structure that *snd_mixer_info()* fills in with the information about the mixer device.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_info()* function fills the *info* structure with information about the mixer device, including the:

- device name
- device type
- number of mixer groups and elements the mixer contains.

Returns:

Zero on success, or a negative value on error.

Errors:

-EINVAL

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_info_t

Information about a mixer

Synopsis:

```
typedef struct snd_mixer_info_s
{
    uint32_t    type;
    uint32_t    attrib;
    uint32_t    elements;
    uint32_t    groups;
    char        id[64];
    char        name[64];
    uint8_t     reserved[128];  /* must be filled with zeroes
 */
}       snd_mixer_info_t;
```

Description:

The snd_mixer_info_t structure describes information about a mixer. You can fill this structure by calling *snd_mixer_info()* (p. 121).

The members include:

type

The sound card type. Deprecated; don't use this member.

attrib

Not used.

elements

The total number of mixer elements in this mixer device.

groups

The total number of mixer groups in this mixer device.

id[64]

The ID of this PCM device (user selectable).

name[64]

The name of the device.

Classification:

QNX Neutrino

snd_mixer_open()

Create a connection and handle to a specified mixer device

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_open( snd_mixer_t **handle,
                    int card,
                    int device );
```

Arguments:

handle

A pointer to a location where *snd_mixer_open()* stores a handle for the mixer device.

card

The card number.

device

The device number.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_open()* function creates a connection and handle to the mixer device specified by the *card* and *device* number. You'll use this handle when calling the other *snd_mixer_** functions.

Returns:

Zero on success, or a negative value on error.

Errors:

-EACCES

Search permission is denied on a component of the path prefix, or the device exists and the permissions specified are denied.

-EINTR

The *open()* operation was interrupted by a signal.

-EMFILE

Too many file descriptors are currently in use by this process.

-ENFILE

Too many files are currently open in the system.

-ENOENT

The named device doesn't exist.

-ENOMEM

No memory available for data structure.

-SND_ERROR_INCOMPATIBLE_VERSION

The audio driver version is incompatible with the client library that the application is using.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_open_name()

Create a connection and handle to a mixer device specified by name

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_open_name( snd_mixer_t **handle,
                         char *name );
```

Arguments:

handle

A pointer to a location where *snd_mixer_open_name()* can store a handle for the mixer device.

name

The full path of the mixer device to open (e.g., `/dev/snd/mixerC0`).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_open_name()* function creates a handle and opens a connection to the named mixer device. You'll use this handle when calling the other *snd_mixer_** functions.

Returns:

Zero on success, or a negative value on error.

Errors:

-EACCES

Search permission is denied on a component of the path prefix, or the device exists and the permissions specified are denied.

-EINTR

The *open()* operation was interrupted by a signal.

-EMFILE

Too many file descriptors are currently in use by this process.

-ENFILE

Too many files are currently open in the system.

-ENOENT

The named device doesn't exist.

-ENOMEM

Not enough memory is available for the data structure.

-SND_ERROR_INCOMPATIBLE_VERSION

The audio driver version is incompatible with the client library that the application is using.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_read()

Read pending mixer events

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_read(
        snd_mixer_t *handle,
        snd_mixer_callbacks_t *callbacks );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

callbacks

A pointer to a `snd_mixer_callbacks_t` (p. 87) structure that defines the list of callbacks.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_read()* function reads pending mixer events from the mixer handle. As each event is read, the list of callbacks is checked for a handler for this event. If a match is found, the callback is invoked. This function is usually called when the *select()* library call indicates that there is data to be read on the mixer's file descriptor.

Returns:

The number of events read from the handle, or a negative value on error.

Errors:

-EBADF

Invalid file descriptor. Your *handle* may be corrupt.

-EINTR

The read operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn't report partial transfers.

-EIO

An event I/O error occurred.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_routes()

Get the number of routes in the mixer and their IDs

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_routes( snd_mixer_t *handle,
                      snd_mixer_routes_t *routes );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

routes

A pointer to a `snd_mixer_routes_t` (p. 132) structure that *snd_mixer_routes()* fills in with information about the routes.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_routes()* function fills the given `snd_mixer_routes_t` structure with the number of routes in the mixer that the handle was opened on. It also fills in the array of route IDs pointed to by *proutes* to a limit of *routes_size*.

We recommend that you work with mixer groups instead of manipulating the elements directly.

Before calling *snd_mixer_routes()*, set the members of this structure as follows:

proutes

This pointer must be `NULL`, or point to a valid storage location for the routes (i.e., an array of `snd_mixer_eid_t` (p. 92) structures).

routes_size

The size of this storage location in `sizeof( snd_mixer_eid_t )` units (i.e., the number of entries in the *proutes* array).

On a successful return, the function sets these members:

routes

The total number of routes in the mixer.

routes_over

The number of routes that couldn't be copied to the storage location.

proutes

The list of routes.

Returns:

Zero on success, or a negative value on error.

Errors:

`-EINVAL`

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_routes_t

Information about mixer routes

Synopsis:

```
typedef struct snd_mixer_routes_s
{
    snd_mixer_eid_t eid;
    int32_t     routes, routes_size, routes_over;
    uint8_t     zero[4];            /* alignment -- zero fill
 */
    snd_mixer_eid_t *proutes;
    void        *pzero;             /* align pointers on
64-bits;
                                       point to NULL */
    uint8_t     reserved[128];      /* must be filled with
zeroes */
}       snd_mixer_routes_t;
```

Description:

The `snd_mixer_routes_t` structure describes all of the routes in a mixer. You can fill this structure by calling *snd_mixer_routes()* (p. 130).

We recommend that you work with mixer groups instead of manipulating the elements directly.

The members of the `snd_mixer_routes_t` structure include:

eid

A pointer to a `snd_mixer_eid_t` (p. 92) structure.

routes

The total number of routes in the mixer.

routes_size

The size of this storage location in `sizeof(snd_mixer_eid_t)` units (i.e., the number of entries in the *proutes* array). Set this member before calling *snd_mixer_routes()*.

routes_over

The number of routes that couldn't be copied to the storage location.

proutes

`NULL`, or an array of `snd_mixer_eid_t` (p. 92) structures.

If *proutes* isn't `NULL`, *snd_mixer_routes()* stores the route IDs in the array.

Classification:

QNX Neutrino

snd_mixer_set_bit()

Set the boolean value of a single bit in the specified bitmap

Synopsis:

```
#include <sys/asoundlib.h>

void snd_mixer_set_bit( unsigned int *bitmap,
                        int bit,
                        int val );
```

Arguments:

bitmap

The bitmap to set. Note that *bitmap* is an array and may be longer than 32 bits.

bit

The index into *bitmap* of the bit to set.

val

The boolean value to store in the bit. Any *value* other than zero causes the bit to be set; a *value* of zero causes it to be cleared.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_set_bit()* function is a convenience function that sets the value (0 or 1) of the bit specified by *bit* in the *bitmap*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Yes

snd_mixer_set_filter()

Set the mask of mixer events that the driver will track

Synopsis:

```
#include <sys/asoundlib.h>

int snd_mixer_set_filter(
        snd_mixer_t *handle,
        snd_mixer_filter_t *filter );
```

Arguments:

handle

The handle for the mixer device. This must have been created by *snd_mixer_open()* (p. 124).

filter

A pointer to a `snd_mixer_filter_t` (p. 104) structure that defines a mask of events to track.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_set_filter()* function uses the `snd_mixer_filter_t` structure to set the mask of all mixer events for the mixer that the handle was opened on that the driver will track. Only those events that are specified in the mask are tracked; all others are discarded as they occur.

You can arrange to have your application receive notification when an event occurs by calling *select()* on the mixer's file descriptor, which you can get by calling *snd_mixer_file_descriptor()* (p. 102). You can use *snd_mixer_read()* (p. 128) to read the event's data.

Returns:

Zero on success, or a negative value on error.

Errors:

-EINVAL

Invalid *handle* or *filter* is `NULL`.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_mixer_sort_eid_table()

Sort a list of element ID structures

Synopsis:

```
#include <sys/asoundlib.h>

void snd_mixer_sort_eid_table(
        snd_mixer_eid_t *list,
        int count,
        snd_mixer_weight_entry_t *table );
```

Arguments:

list

A pointer to the list of `snd_mixer_eid_t` (p. 92), structures that you want to sort.

count

The number of entries in the list.

table

A pointer to an array of `snd_mixer_weight_entry_t` (p. 142) structures that defines the relative weights for the elements.

Most applications use the default table weight structure, *snd_mixer_default_weights*.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_sort_eid_table()* function sorts a list of `eid` (element id structures) based on the names and the relative weights specified by the weight table.

We recommend that you work with mixer groups instead of manipulating the elements directly.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

snd_mixer_sort_gid_table()

Sort a list of group ID structures

Synopsis:

```
#include <sys/asoundlib.h>

void snd_mixer_sort_gid_table(
        snd_mixer_gid_t *list,
        int count,
        snd_mixer_weight_entry_t *table );
```

Arguments:

list

The list of `snd_mixer_gid_t` (p. 110) structures that you want to sort.

count

The number of entries in the list.

table

A pointer to an array of `snd_mixer_weight_entry_t` (p. 142) structures that defines the relative weights for the groups.

Most applications use the default table weight structure, *snd_mixer_default_weights*.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_mixer_sort_gid_table()* function sorts a list of `gid` (group id structures) based on the names and the relative weights specified by the weight table.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

snd_mixer_weight_entry_t

Weight table for sorting mixer element and group IDs

Synopsis:

```
typedef struct {
    char *name;
    int weight;
} snd_mixer_weight_entry_t;
```

Description:

The `snd_mixer_weight_entry_t` structure defines the weights that *snd_mixer_sort_eid_table()* (p. 138) and *snd_mixer_sort_gid_table()* (p. 140) use to sort mixer element and group IDs. The members include:

name

The name of the mixer element or group.

weight

The weight to use when sorting.

Classification:

QNX Neutrino

snd_pcm_build_linear_format()

Encode a linear format value

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_build_linear_format( int width,
                                 int unsigned,
                                 int big_endian );
```

Arguments:

width

The width; one of 8, 16, 24, or 32.

unsigned

0 for signed; 1 for unsigned.

big_endian

0 for little endian; 1 for big endian.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_build_linear_format()* function returns the linear format value encoded from the given components. For a list of the supported linear formats, see *snd_pcm_format_linear()* (p. 196).

Returns:

A positive value (`SND_PCM_SFMT_*`) on success, or -1 if the arguments are invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_capture_flush()

Discard all pending data in a PCM capture channel's queue and stop the channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_capture_flush( snd_pcm_t *handle);
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_capture_flush()* function throws away all unprocessed data in the driver queue.

If the operation is successful (zero is returned), the channel's state is changed to `SND_PCM_STATUS_READY`, and the channel is stopped.

This function *isn't* plugin-aware. It functions exactly the same way as `snd_pcm_channel_flush(.., SND_PCM_CHANNEL_CAPTURE)`. Make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

Returns:

Zero on success, or a negative error code.

Errors:

-EBADFD

The pcm device state isn't ready.

-EINTR

The driver isn't processing the data (Internal Error).

-EINVAL

Invalid *handle*.

-EIO

An invalid channel was specified, or the data wasn't all flushed.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_capture_go()

Start a PCM capture channel running

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_capture_go ( snd_pcm_t *handle );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Description:

The *snd_pcm_capture_go()* function starts the capture channel.

You should call this function only when the driver is in the `SND_PCM_STATUS_READY` state. Calling this function is required if you've set your capture channel's start state to `SND_PCM_START_GO` (see *snd_pcm_plugin_params()* (p. 247)). You can also use this function to "kick start" early a capture channel that has a start state of `SND_PCM_START_DATA` or `SND_PCM_START_FULL`.

If the parameters are valid (i.e., *snd_pcm_capture_go()* returns zero), then the driver state is changed to `SND_PCM_STATUS_RUNNING`.

This function is safe to use with plugin-aware functions.

This call is used identically to *snd_pcm_plugin_params()* (p. 247).

Returns:

`EOK`

Success.

`-EINVAL`

Invalid *handle*.

-EIO

Invalid *channel*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_capture_pause()

Pause a channel that's capturing

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_capture_pause ( snd_pcm_t *pcm );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_capture_pause()* function pauses a channel that's capturing. Unlike draining or flushing, this preserves all data that has not yet been received within the audio driver, to be retrieved after resuming.

Returns:

EOK

Success.

-EINVAL

The handle is `NULL`, or the channel isn't capturing.

This function can return other negative *errno* values.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_capture_prepare()

Signal the driver to ready the capture channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_capture_prepare( snd_pcm_t *handle);
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_capture_prepare()* function prepares hardware to operate in a specified transfer direction. This call is responsible for all parts of the hardware's startup sequence that require additional initialization time, allowing the final "GO" (either from writes into the buffers or *snd_pcm_channel_go()* (p. 157)) to execute more quickly.

You can call this function in all states except `SND_PCM_STATUS_NOTREADY` (returns `-EBADFD`) and `SND_PCM_STATUS_RUNNING` state (returns `-EBUSY`). If the operation is successful (zero is returned), the driver state is changed to `SND_PCM_STATUS_PREPARED`.

If your channel has overrun, you have to reprepare it before continuing. For an example, see ***`waverec.c`*** *example* in the appendix.

Returns:

Zero on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle*.

-EBUSY

Channel is running.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_capture_resume()

Resume a channel that was paused while capturing

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_capture_resume ( snd_pcm_t *pcm );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_capture_resume()* function resumes a channel that was paused while capturing.

Returns:

EOK

Success.

-EINVAL

The handle is `NULL`, or the channel wasn't being used for capturing.

This function can return other negative *errno* values.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_flush()

Flush all pending data in a PCM channel's queue and stop the channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_flush( snd_pcm_t *handle,
                           int channel );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel; `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_flush()* function flushes all unprocessed data in the driver queue by calling *snd_pcm_capture_flush()* (p. 145) or *snd_pcm_playback_flush()* (p. 231), depending on the value of *channel*.

Returns:

Zero on success, or a negative error code.

Errors:

-EBADFD

The PCM device state isn't Ready.

-EINTR

The driver isn't processing the data (Internal Error).

-EINVAL

Invalid *handle*.

-EIO

An invalid channel was specified, or the data wasn't all flushed.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_go()

Start a PCM channel running

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_go ( snd_pcm_t *handle,
                         int channel );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel; `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

Library:

`libasound.so`

Description:

The *snd_pcm_channel_go()* function starts the channel running by calling *snd_pcm_capture_go()* (p. 147) or *snd_pcm_playback_go()* (p. 233), depending on the value of *channel*.

The function should be called in `SND_PCM_STATUS_READY` state. Calling this function is required if you've set your channel's start state to `SND_PCM_START_GO` (see *snd_pcm_plugin_params()* (p. 247)). You can also use this function to “kick start” early a channel that has a start state of `SND_PCM_START_DATA` or `SND_PCM_START_FULL`.

When you're using *snd_pcm_channel_go()* for playback, ensure that two or more audio fragments have been written into the audio interface before issuing the go command, to prevent the audio channel/stream from going into the UNDERRUN state.

If the parameters are valid (i.e., the function returns zero), then the driver state is changed to `SND_PCM_STATUS_RUNNING`.

This function is safe to use with plugin-aware functions. This call is used identically to *snd_pcm_plugin_params()* (p. 247).

Returns:

EOK

Success.

-EINVAL

Invalid *handle*.

-EIO

Invalid *channel*.

-EMORE

Insufficient audio fragments have been written to the audio interface (when writing to the software mixer device).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_info()

Get information about a PCM channel's current capabilities

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_info(
        snd_pcm_t *handle,
        snd_pcm_channel_info_t *info );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

info

A pointer to a `snd_pcm_channel_info_t` (p. 161) structure that *snd_pcm_channel_info()* fills with information about the PCM channel.

Before calling this function, set the *info* structure's *channel* member to specify the direction. This function sets all the other members.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_channel_info()* function fills the *info* structure with the current capabilities of the PCM channel selected by *handle*.

This function and the plugin-aware version, *snd_pcm_plugin_info()* (p. 245), get a dynamic "snapshot" of the system's current capabilities, which can shrink and grow as subchannels are allocated and freed. They're similar to *snd_ctl_pcm_channel_info()* (p. 81), which gets information about the *complete* capabilities of the system.

Returns:

Zero on success, or a negative error code.

Errors:

-EINVAL

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_info_t

Information structure for a PCM channel

Synopsis:

```
typedef struct snd_pcm_channel_info
{
    int32_t     subdevice;
    int8_t      subname[36];
    int32_t     channel;
    int32_t     zero1;
    int32_t     zero2[4];
    uint32_t    flags;
    uint32_t    formats;
    uint32_t    rates;
    int32_t     min_rate;
    int32_t     max_rate;
    int32_t     min_voices;
    int32_t     max_voices;
    int32_t     max_buffer_size;
    int32_t     min_fragment_size;
    int32_t     max_fragment_size;
    int32_t     fragment_align;
    int32_t     fifo_size;
    int32_t     transfer_block_size;
    uint8_t     zero3[4];

    snd_pcm_digital_t dig_mask;
    uint32_t          zero4;
    int32_t           mixer_device;
    snd_mixer_eid_t   mixer_eid;
    snd_mixer_gid_t   mixer_gid;
    uint8_t           reserved[128];
}       snd_pcm_channel_info_t;
```

Description:

The `snd_pcm_channel_info_t` structure describes PCM channel information. The members include:

subdevice

The subdevice number.

***subname*[32]**

The subdevice name.

channel

The channel direction; either `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

flags

Any combination of:

- SND_PCM_CHNINFO_BLOCK — the hardware supports block mode.
- SND_PCM_CHNINFO_BLOCK_TRANSFER — the hardware transfers samples by chunks (for example PCI burst transfers).
- SND_PCM_CHNINFO_INTERLEAVE — the hardware accepts audio data composed of interleaved samples.
- SND_PCM_CHNINFO_MMAP — the hardware supports mmap access.
- SND_PCM_CHNINFO_MMAP_VALID — fragment samples are valid during transfer. This means that the fragment samples may be used when the *io* member from the **mmap** control structure snd_pcm_mmap_control_t is set (the fragment is being transferred).
- SND_PCM_CHNINFO_NONINTERLEAVE — the hardware accepts audio data composed of noninterleaved samples.
- SND_PCM_CHNINFO_OVERRANGE — the hardware supports ADC (capture) overrange detection.
- SND_PCM_CHNINFO_PAUSE — the hardware supports pausing of the DMA engines (playback only).

Note that the absence of this flag does not preclude the synthesis of an application-level pause. It refers only to the direct capabilities of the hardware. Support for this flag is extremely rare, so dependence on it is discouraged.

formats

The supported formats (SND_PCM_FMT_*).

rates

Hardware rates (SND_PCM_RATE_*).

min_rate

The minimum rate (in Hz).

max_rate

The maximum rate (in Hz).

min_voices

The minimum number of voices (probably always 1).

max_voices

The maximum number of voices.

max_buffer_size

The maximum buffer size, in bytes.

min_fragment_size

The minimum fragment size, in bytes.

max_fragment_size

The maximum fragment size, in bytes.

fragment_align

If this value is set, the size of the buffer fragments must be a multiple of this value, so that they are in the proper alignment.

fifo_size

The stream FIFO size, in bytes. Deprecated; don't use this member.

transfer_block_size

The bus transfer block size in bytes.

dig_mask

Not currently implemented.

mixer_device

The mixer device for this channel.

mixer_eid

A `snd_mixer_eid_t` (p. 92) structure that describes the mixer element identification for this channel.

mixer_gid

The mixer group identification for this channel; see `snd_mixer_gid_t` (p. 110). You should use this mixer group in applications that are implementing their own volume controls.

This mixer group is guaranteed to be the lowest-level mixer group for your channel (or subchannel), as determined at the time that you call *snd_ctl_pcm_channel_info()* (p. 81). If you call this function after the channel has been configured, and a subchannel has been allocated (i.e., after calling *snd_pcm_channel_params()* (p. 165)), this mixer group is the subchannel mixer group that's specific to the application's current subchannel.

Classification:

QNX Neutrino

snd_pcm_channel_params()

Set a PCM channel's configurable parameters

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_params(
        snd_pcm_t *handle,
        snd_pcm_channel_params_t *params );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

params

A pointer to a `snd_pcm_channel_params_t` (p. 167) structure in which you've specified the PCM channel's configurable parameters. All members are write-only.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_channel_params()* function sets up the transfer parameters according to the *params* structure.

You can call the function in `SND_PCM_STATUS_NOTREADY` (initial) and `SND_PCM_STATUS_READY` states; otherwise, *snd_pcm_channel_params()* returns `-EBADFD`.

If the parameters are valid (i.e., *snd_pcm_channel_params()* returns zero), the driver state is changed to `SND_PCM_STATUS_READY`.

The ability to convert audio to match hardware capabilities (for example, voice conversion, rate conversion, type conversion, etc.) is enabled by default. As a result, this function behaves as *snd_pcm_plugin_params()* (p. 247), unless you've disabled the conversion by calling:

```
snd_pcm_plugin_set_disable(handle, PLUGIN_CONVERSION);
```

Returns:

EOK

Success.

-EINVAL

Invalid *handle* or *params*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_params_t

PCM channel parameters

Synopsis:

```
typedef struct snd_pcm_channel_params
{
    int32_t             channel;
    int32_t             mode;
    snd_pcm_sync_t      sync;               /* hardware synchronization ID */

    snd_pcm_format_t    format;
    snd_pcm_digital_t   digital;
    int32_t             start_mode;
    int32_t             stop_mode;
    int32_t             time:1, ust_time:1;
    uint32_t            why_failed;         /* SND_PCM_PARAMS_BAD_??? */
    union
    {
        struct
        {
            int32_t     queue_size;
            int32_t     fill;
            int32_t     max_fill;
            uint8_t     reserved[124];      /* must be filled with zeroes */
        }       stream;
        struct
        {
            int32_t     frag_size;
            int32_t     frags_min;
            int32_t     frags_max;
            uint32_t    frags_buffered_max;
            uint8_t     reserved[120];      /* must be filled with zeroes */
        }       block;
        uint8_t     reserved[128];      /* must be filled with zeroes */
    }       buf;
    char        sw_mixer_subchn_name[32];
    uint8_t     reserved[96];         /* must be filled with zeroes */
} snd_pcm_channel_params_t;
```

Description:

The `snd_pcm_channel_params_t` structure describes the parameters of a PCM capture or playback channel. The members include:

channel

The channel direction; one of `SND_PCM_CHANNEL_PLAYBACK` or `SND_PCM_CHANNEL_CAPTURE`.

mode

The channel mode: `SND_PCM_MODE_BLOCK`. (`SND_PCM_MODE_STREAM` is deprecated.)

You can OR in the following flags:

- `SND_PCM_MODE_FLAG_PROTECTED_CONTENT` — indicates that the output path may not change without the client's approval. When the output path does change, the client will change to state SND_PCM_STATUS_UNSECURE. The client can then check the current output path, and call *snd_pcm_channel_prepare()* (p. 173) to continue playback.

- SND_PCM_MODE_FLAG_REQUIRE_PROTECTION — indicates that digital protection is required on the audio path. For example, for HDMI, it indicates that HDCP must be enabled for audio to play.

format

The data format; see `snd_pcm_format_t` (p. 204).

digital

Not currently implemented.

start_mode

The start mode; one of:

- SND_PCM_START_DATA — start when some data is written (playback) or requested (capture).
- SND_PCM_START_FULL — start when the whole queue is filled (playback only).
- SND_PCM_START_GO — start on the Go command.

stop_mode

The stop mode; one of:

- SND_PCM_STOP_STOP — stop when an underrun or overrun occurs.
- SND_PCM_STOP_ERASE — stop and erase the whole buffer when an overrun occurs (capture only).
- SND_PCM_STOP_ROLLOVER — ROLLOVER (i.e. automatically reprepare and continue) when an underrun or overrun occurs.

time

If set, the driver offers, in the status structure, the time when the transfer began. The time is in the format used by *gettimeofday()* (see the QNX Neutrino *C Library Reference*).

ust_time

If set, the driver offers, in the status structure, the time when the transfer began. The time is in UST format.

sync

The synchronization group. Not supported; don't use this member.

queue_size

The queue size, in bytes, for the stream mode. Not supported; don't use this member.

fill

The fill mode (`SND_PCM_FILL_*` constants). Not supported; don't use this member.

max_fill

The number of bytes to be filled ahead with silence. Not supported; don't use this member.

frag_size

The size of fragment, in bytes.

frags_min

Depends on the mode:

- Capture — the minimum filled fragments to allow wakeup (usually one).
- Playback — the minimum free fragments to allow wakeup (usually one).

frags_max

For playback, the maximum filled fragments to allow wakeup. This value specifies the total number of fragments that could be written to by an application. This excludes the fragment that's currently playing, so the actual total number of fragments is *frags_max* + 1.

frags_buffered_max

If this is set, `io-audio` may block the caller after fewer than *frags_max* fragments have been passed, if it chooses, but won't block the client before *frags_buffered_max* fragments have been written.

sw_mixer_subchn_name

By default, the name of all sw_mixer subchannel groups is `PCM Subchannel`; you can use this field to assign a different name.

Classification:

QNX Neutrino

snd_pcm_channel_pause()

Pause a channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_pause ( snd_pcm_t *pcm,
                            int channel );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel; `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_channel_pause()* function pauses a channel by calling *snd_pcm_capture_pause()* (p. 149) or *snd_pcm_playback_pause()* (p. 235), depending on the value of *channel*.

Unlike draining or flushing, this preserves all data that has not yet been received or played out within the audio driver, to be retrieved or played out after resuming.

Returns:

EOK

Success.

-EINVAL

The handle is `NULL`.

-EIO

The channel isn't valid.

This function can return other negative *errno* values.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_prepare()

Signal the driver to ready the specified channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_prepare( snd_pcm_t *handle,
                             int channel );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_channel_prepare()* function prepares hardware to operate in a specified transfer direction by calling *snd_pcm_capture_prepare()* (p. 151) or *snd_pcm_playback_prepare()* (p. 237), depending on the value of *channel*.

This call is responsible for all parts of the hardware's startup sequence that require additional initialization time, allowing the final “GO” (either from writes into the buffers or *snd_pcm_channel_go()* (p. 157)) to execute more quickly.

This function may be called in all states except SND_PCM_STATUS_NOTREADY (returns -EBADFD) and SND_PCM_STATUS_RUNNING state (returns -EBUSY). If the operation is successful (zero is returned), the driver state is changed to SND_PCM_STATUS_PREPARED.

If your channel has underrun (during playback) or overrun (during capture), you have to reprepare it before continuing. For an example, see `wave.c` and `waverec.c` in the appendix.

Returns:

Zero on success, or a negative error code.

Errors:

-EINVAL

Invalid *handle*.

-EBUSY

Channel is running.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_resume()

Resume a channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_resume ( snd_pcm_t *pcm,
                             int channel );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel; `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_channel_resume()* function resumes a channel by calling *snd_pcm_capture_resume()* (p. 153) or *snd_pcm_playback_resume()* (p. 239), depending on the value of *channel*.

Returns:

EOK

Success.

-EINVAL

The handle is `NULL`.

-EIO

The channel isn't valid.

This function can return other negative *errno* values.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_setup()

Get the current configuration for the specified PCM channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_setup(
        snd_pcm_t *handle,
        snd_pcm_channel_setup_t *setup );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

setup

A pointer to a `snd_pcm_channel_setup_t` (p. 179) structure that *snd_pcm_channel_setup()* fills with information about the PCM channel setup.

Set the *setup* structure's *channel* member to specify the direction. All other members are read-only.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_channel_setup()* function fills the *setup* structure with data about the PCM channel's configuration.

The ability to convert audio to match hardware capabilities (for example, voice conversion, rate conversion, type conversion, etc.) is enabled by default. As a result, this function behaves as *snd_pcm_plugin_setup()* (p. 266), unless you've disabled the conversion by calling:

```
snd_pcm_plugin_set_disable(handle, PLUGIN_CONVERSION);
```

Returns:

EOK

Success.

-EINVAL

Invalid *handle*

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_setup_t

Current configuration of a PCM channel

Synopsis:

```
typedef struct snd_pcm_channel_setup
{
    int32_t             channel;
    int32_t             mode;
    snd_pcm_format_t    format;
    snd_pcm_digital_t   digital;
    union
    {
        struct
        {
            int32_t     queue_size;
            uint8_t     reserved[124]; /* must be filled with zeroes */
        }       stream;
        struct
        {
            int32_t     frag_size;
            int32_t     frags;
            int32_t     frags_min;
            int32_t     frags_max;
            uint32_t    max_frag_size;
            uint8_t     reserved[124]; /* must be filled with zeroes */
        }       block;
        uint8_t     reserved[128];     /* must be filled with zeroes */
    }       buf;
    int16_t         msbits_per_sample;
    int16_t         pad1;
    int32_t         mixer_device;
    snd_mixer_eid_t *mixer_eid;
    snd_mixer_gid_t *mixer_gid;
    uint8_t         mmap_valid:1;
    uint8_t         mmap_active:1;
    int32_t         mixer_card;
    uint8_t         reserved[104];     /* must be filled with zeroes */
}       snd_pcm_channel_setup_t;
```

Description:

The `snd_pcm_channel_setup_t` structure describes the current configuration of a PCM channel. The members include:

channel

The channel direction; One of SND_PCM_CHANNEL_PLAYBACK or SND_PCM_CHANNEL_CAPTURE.

mode

The channel mode: SND_PCM_MODE_BLOCK. (SND_PCM_MODE_STREAM is deprecated.)

format

The data format; see `snd_pcm_format_t` (p. 204). Note that the *rate* member may differ from the requested one.

digital

Not currently implemented.

queue_size

The real queue size (which may differ from requested one).

frag_size

The real fragment size (which may differ from requested one).

frags

The number of fragments.

frags_min

Capture: the minimum filled fragments to allow wakeup. Playback: the minimum free fragments to allow wakeup.

frags_max

Playback: the maximum filled fragments to allow wakeup. The value also specifies the maximum number of used fragments plus one.

max_frag_size

The maximum fragment size.

msbits_per_sample

How many most-significant bits are physically used.

mixer_device

Mixer device for this subchannel.

mixer_eid

A pointer to the mixer element identification for this subchannel.

mixer_gid

A pointer to the mixer group identification for this subchannel; see `snd_mixer_gid_t` (p. 110).

mmap_valid

The channel can use mmapped access.

mmap_active

The channel is using mmapped transfers.

mixer_card

The mixer card.

Classification:

QNX Neutrino

snd_pcm_channel_status()

Get the runtime status of a PCM channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_channel_status(
        snd_pcm_t *handle,
        snd_pcm_channel_status_t *status );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

status

A pointer to a `snd_pcm_channel_status_t` (p. 184) structure that *snd_pcm_channel_status()* fills with information about the PCM channel's status.

Fill in the *status* structure's *channel* member to specify the direction. All other members are read-only.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_channel_status()* function fills the *status* structure with data about the PCM channel's runtime status.

The ability to convert audio to match hardware capabilities (for example, voice conversion, rate conversion, type conversion, etc.) is enabled by default. As a result, this function behaves as *snd_pcm_plugin_status()* (p. 270), unless you've disabled the conversion by calling:

```
snd_pcm_plugin_set_disable(handle, PLUGIN_CONVERSION);
```

Returns:

EOK

Success.

-EBADFD

The PCM device state isn't Ready.

-EFAULT

Failed to copy data.

-EINVAL

Invalid *handle* or data pointer is `NULL`.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_channel_status_t

PCM channel status structure

Synopsis:

```
typedef struct snd_pcm_channel_status
{
    int32_t                  channel;
    int32_t                  mode;
    int32_t                  status;
    uint32_t                 scount;
    struct timeval           stime;
    uint64_t                 ust_stime;
    int32_t                  frag;
    int32_t                  count;
    int32_t                  free;
    int32_t                  underrun;
    int32_t                  overrun;
    int32_t                  overrange;
    uint32_t                 subbuffered;
    snd_pcm_status_data_t    status_data;
    struct timeval           stop_time;
    uint8_t                  hw_device;
    uint8_t                  reserved[111];  /* must be filled
 with zero */
} snd_pcm_channel_status_t;
```

Description:

The `snd_pcm_channel_status_t` structure describes the status of a PCM channel. The members include:

channel

The channel direction; one of `SND_PCM_CHANNEL_PLAYBACK` or `SND_PCM_CHANNEL_CAPTURE`.

mode

The transfer mode: `SND_PCM_MODE_BLOCK`. (`SND_PCM_MODE_STREAM` is deprecated.)

status

The channel status. Valid values are:

- `SND_PCM_STATUS_NOTREADY` — the driver isn't prepared for any operation. After a successful call to *snd_pcm_channel_params()* (p. 165), the state is changed to `SND_PCM_STATUS_READY`.

- `SND_PCM_STATUS_READY` — the driver is ready for operation. You can *mmap()* the audio buffer only in this state, but the samples still can't be transferred. After a successful call to *snd_pcm_channel_prepare()* (p. 173), *snd_pcm_capture_prepare()* (p. 151), *snd_pcm_playback_prepare()* (p. 237), or *snd_pcm_plugin_prepare()* (p. 251), the state is changed to `SND_PCM_STATUS_PREPARED`.
- `SND_PCM_STATUS_PREPARED` — the driver is prepared for operation. The samples may be transferred in this state.
- `SND_PCM_STATUS_RUNNING` — the driver is actively transferring data through the hardware. The samples may be transferred in this state.
- `SND_PCM_STATUS_UNDERRUN` — the playback channel is in an underrun state. The driver completely drained the buffers before new data was ready to be played. You must reprepare the channel before continuing, by calling *snd_pcm_channel_prepare()* (p. 173), *snd_pcm_playback_prepare()* (p. 237), or *snd_pcm_plugin_prepare()* (p. 251). See the *`wave.c` example* in the appendix.
- `SND_PCM_STATUS_OVERRUN` — the capture channel is in an overrun state. The driver has processed the incoming data faster than it's coming in; the channel is stalled. You must reprepare the channel before continuing, by calling *snd_pcm_channel_prepare()* (p. 173), *snd_pcm_capture_prepare()* (p. 151), or *snd_pcm_plugin_prepare()* (p. 251). See the *`waverec.c` example* in the appendix.
- `SND_PCM_STATUS_PAUSED` — the playback is paused (not supported by QSA).

scount

The number of bytes processed since the playback/capture last started. This value wraps around to 0 when it passes the value of `SND_PCM_BOUNDARY`, and is reset when you prepare the channel.

stime

The playback/capture start time, in the format used by *gettimeofday()* (see the QNX Neutrino *C Library Reference*).

This member is valid only when the *time* flags is active in the `snd_pcm_channel_params_t` (p. 167) structure.

ust_stime

The playback/capture start time, in UST format. This member is valid only when the *ust_time* flags is active in the `snd_pcm_channel_params_t` (p. 167) structure.

frag

The current fragment number (available only in the block mode).

count

The number of bytes in the queue/buffer; see the note below.

free

The number of bytes in the queue that are still free; see the note below.

underrun

The number of playback underruns since the last status.

overrun

The number of capture overruns since the last status.

overrange

The number of ADC capture overrange detections since the last status.

subbuffered

The number of bytes subbuffered in the plugin interface.

status_data

Additional data relevant to a particular value for *status*.

stop_time

The time at which the last sample was played or recorded in the most recent case where the channel stopped for any reason.

hw_device

Which device, from the `audio_manager_device_t` enumerated type, is currently being used. It might not be available, in which case, it's set to `AUDIO_DEVICE_COUNT`.

- The *count* and *free* members aren't used if the mmap plugin is used. To disable the mmap plugin, call *snd_pcm_plugin_set_disable()* (p. 256).
- The *stop_time* and *stime* members hold valid data only if there's data present in the channel; they aren't necessarily valid as soon as the channel goes to a `SND_PCM_STATUS_RUNNING` state, but they're guaranteed to be valid as soon as *scount* indicates that some data has been processed.

Classification:

QNX Neutrino

snd_pcm_close()

Close a PCM handle and free its resources

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_close( snd_pcm_t *handle );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_close()* function frees all resources allocated with the audio handle and closes the connection to the PCM interface.

Returns:

Zero on success, or a negative value on error.

Errors:

`-EINTR`

The *close()* call was interrupted by a signal.

`-EINVAL`

Invalid *handle* argument.

`-EIO`

An I/O error occurred while updating the directory information.

`-ENOSPC`

A previous buffered write call has failed.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_file_descriptor()

Return the file descriptor of the connection to the PCM interface

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_file_descriptor( snd_pcm_t *handle,
                             int channel );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel; `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_file_descriptor()* function returns the file descriptor of the connection to the PCM interface.

You can use this file descriptor for the *select()* synchronous multiplexer function (see the QNX Neutrino *C Library Reference*).

Returns:

The file descriptor of the connection to the PCM interface on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle* argument.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_find()

Find all PCM devices in the system that meet the given criteria

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_find( unsigned int format,
                  int *number,
                  int *cards,
                  int *devices,
                  int mode );
```

Arguments:

format

Any combination of the SND_PCM_FMT_* constants. Here are the most commonly used flags:

- SND_PCM_FMT_U8 — unsigned 8-bit PCM.
- SND_PCM_FMT_S8 — signed 8-bit PCM.
- SND_PCM_FMT_U16_LE — unsigned 16-bit PCM little endian.
- SND_PCM_FMT_U16_BE — unsigned 16-bit PCM big endian.
- SND_PCM_FMT_S16_LE — signed 16-bit PCM little endian.
- SND_PCM_FMT_S16_BE — signed 16-bit PCM big endian.

number

The size of the card and device arrays that *cards* and *devices* point to. On return, *number* contains the total number of devices found.

cards

An array in which *snd_pcm_find()* stores the numbers of the cards it finds.

devices

An array in which *snd_pcm_find()* stores the numbers of the devices it finds.

mode

One of the following:

- SND_PCM_CHANNEL_PLAYBACK — the playback channel.
- SND_PCM_CHANNEL_CAPTURE — the capture channel.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_find()* function finds all PCM devices in the system that support any combination of the given *format* parameters in the given *mode*.

The card and device arrays are to be considered paired: the following uniquely defines the first PCM device:

```
card[0] + device[0]
```

Returns:

A positive integer representing the total number of devices found (same as *number* on return), or a negative value on error.

Errors:

-EINVAL

Invalid *mode* or *format*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_format_big_endian()

Check for a big-endian format

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_format_big_endian( int format );
```

Arguments:

format

The format number (one of the `SND_PCM_SFMT_*` constants). For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_format_big_endian()* function checks to see if *format* is big-endian.

Returns:

1

The format is in big-endian byte order.

0

The format isn't in big-endian byte order.

Otherwise, it returns a negative error code.

Errors:

-EINVAL

Invalid *format* with respect to endianness.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_format_linear()

Check for a linear format

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_format_linear( int format );
```

Arguments:

format

The format number (one of the `SND_PCM_SFMT_*` constants). For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_format_linear()* function checks to see if the format is linear. The supported linear formats are:

- `SND_PCM_SFMT_S8`
- `SND_PCM_SFMT_U8`
- `SND_PCM_SFMT_S16_LE`
- `SND_PCM_SFMT_U16_LE`
- `SND_PCM_SFMT_S16_BE`
- `SND_PCM_SFMT_U16_BE`
- `SND_PCM_SFMT_S24_LE`
- `SND_PCM_SFMT_U24_LE`
- `SND_PCM_SFMT_S24_BE`
- `SND_PCM_SFMT_U24_BE`
- `SND_PCM_SFMT_S32_LE`
- `SND_PCM_SFMT_U32_LE`
- `SND_PCM_SFMT_S32_BE`
- `SND_PCM_SFMT_U32_BE`

For a list of all the supported formats, see *snd_pcm_get_format_name()* (p. 211).

Returns:

1

The format is a linear format.

0

The format isn't a linear format.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_format_little_endian()

Check for a little-endian format

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_format_little_endian( int format );
```

Arguments:

format

The format number (one of the `SND_PCM_SFMT_*` constants). For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_format_little_endian()* function checks to see if *format* is little-endian.

Returns:

1

The format is in little-endian byte order.

0

The format isn't in little-endian byte order.

Otherwise, it returns a negative error code.

Errors:

-EINVAL

Invalid *format* with respect to endianness.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_format_signed()

Check for a signed format

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_format_signed( int format );
```

Arguments:

format

The format number (one of the `SND_PCM_SFMT_*` constants). For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_format_signed()* function checks for a signed format.

Returns:

1

The format is signed.

0

The format is unsigned.

Otherwise, it returns a negative error code.

Errors:

`-EINVAL`

Invalid *format* with respect to sign.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_format_size()

Convert the size in the given samples to bytes

Synopsis:

```
#include <sys/asoundlib.h>

ssize_t snd_pcm_format_size( int format,
                             size_t num_samples );
```

Arguments:

format

The format number (one of the `SND_PCM_SFMT_*` constants). For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211).

num_samples

The number of samples for which you want to determine the size.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_format_size()* function calculates the size, in bytes, of *num_samples* samples of data in the given format.

Returns:

A positive value on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *format*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_format_t

PCM data format structure

Synopsis:

```
typedef struct snd_pcm_format {
    int32_t  interleave: 1;
    int32_t  format;
    int32_t  rate;
    int32_t  voices;
    int32_t  special;
    uint8_t  reserved[124];  /* must be filled with zeroes */
} snd_pcm_format_t;
```

Description:

The `snd_pcm_format_t` structure describes the format of the PCM data. The members include:

interleave

If set, the sample data contains interleaved samples.

format

The format number (one of the `SND_PCM_SFMT_*` constants). For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211).

rate

The requested rate, in Hz.

voices

The number of voices, in the range specified by the *min_voices* and *max_voices* members of the `snd_pcm_channel_info_t` (p. 161) structure. Typical values are 2 for stereo, and 1 for mono.

special

Special (custom) description of format. Use when `SND_PCM_SFMT_SPECIAL` is specified.

Classification:

QNX Neutrino

snd_pcm_format_unsigned()

Check for an unsigned format

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_format_unsigned( int format );
```

Arguments:

format

The format number (one of the `SND_PCM_SFMT_*` constants). For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_format_unsigned()* function checks for an unsigned format.

Returns:

1

The format is unsigned.

0

The format is signed.

Otherwise, it returns a negative error code.

Errors:

`-EINVAL`

Invalid *format* with respect to sign.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_format_width()

Return the sample width in bits for a format

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_format_width( int format );
```

Arguments:

format

The format number (one of the `SND_PCM_SFMT_*` constants). For a list of the supported formats, see *snd_pcm_get_format_name()* (p. 211).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_format_width()* function returns the sample width in bits.

Returns:

A positive sample width on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *format*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

snd_pcm_get_audioman_handle()

Retrieve an audioman handle that's bound to a PCM stream

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_get_audioman_handle( snd_pcm_t *handle,
                                 unsigned int *audioman_handle
 );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

audioman_handle

A pointer to a location where the function can store the handle of the audio manager.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_get_audioman_handle()* function retrieves an audioman handle that's bound to a PCM stream. Binding an audioman handle to a PCM stream results in the PCM stream's conditionally ducking behind other streams, depending on the type of other stream playing.

Returns:

`EOK`

Success.

`-EINVAL`

The PCM handle is `NULL`, or the value of the audio handle passed to the function was `SND_PCM_AUDIOMAN_NO_HANDLE`.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_get_format_name()

Convert a format value into a human-readable text string

Synopsis:

```
#include <sys/asoundlib.h>

const char *snd_pcm_get_format_name( int format );
```

Arguments:

format

The format number (one of the `SND_PCM_SFMT_*` constants).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_get_format_name()* function converts a format member or manifest of the form `SND_PCM_SFMT_*` to a text string suitable for displaying to a human user:

`SND_PCM_SFMT_U8`

Unsigned 8-bit

`SND_PCM_SFMT_S8`

Signed 8-bit

`SND_PCM_SFMT_U16_LE`

Unsigned 16-bit Little Endian

`SND_PCM_SFMT_U16_BE`

Unsigned 16-bit Big Endian

`SND_PCM_SFMT_S16_LE`

Signed 16-bit Little Endian

`SND_PCM_SFMT_S16_BE`

Signed 16-bit Big Endian

SND_PCM_SFMT_U24_LE

Unsigned 24-bit Little Endian

SND_PCM_SFMT_U24_BE

Unsigned 24-bit Big Endian

SND_PCM_SFMT_S24_LE

Signed 24-bit Little Endian

SND_PCM_SFMT_S24_BE

Signed 24-bit Big Endian

SND_PCM_SFMT_U32_LE

Unsigned 32-bit Little Endian

SND_PCM_SFMT_U32_BE

Unsigned 32-bit Big Endian

SND_PCM_SFMT_S32_LE

Signed 32-bit Little Endian

SND_PCM_SFMT_S32_BE

Signed 32-bit Big Endian

SND_PCM_SFMT_A_LAW

A-Law

SND_PCM_SFMT_MU_LAW

Mu-Law

SND_PCM_SFMT_FLOAT_LE

Float Little Endian

SND_PCM_SFMT_FLOAT_BE

Float Big Endian

SND_PCM_SFMT_FLOAT64_LE

Float64 Little Endian

SND_PCM_SFMT_FLOAT64_BE

Float64 Big Endian

SND_PCM_SFMT_IEC958_SUBFRAME_LE

IEC-958 Little Endian

SND_PCM_SFMT_IEC958_SUBFRAME_BE

IEC-958 Big Endian

SND_PCM_SFMT_IMA_ADPCM

Ima-ADPCM

SND_PCM_SFMT_GSM

GSM

SND_PCM_SFMT_MPEG

MPEG

SND_PCM_SFMT_SPECIAL

Special

Returns:

A character pointer to the text format name.

Don't modify the strings that this function returns.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_info()

Get general information about a PCM device

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_info( snd_pcm_t *handle,
                  snd_pcm_info_t *info );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

info

A pointer to a `snd_pcm_info_t` (p. 216) structure in which *snd_ctl_pcm_info()* stores the information.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_info()* function fills the *info* structure with information about the capabilities of the PCM device selected by *handle*.

Returns:

Zero on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_info_t

Capability information about a PCM device

Synopsis:

```
typedef struct snd_pcm_info
{
    uint32_t    type;
    uint32_t    flags;
    uint8_t     id[64];
    char        name[80];
    int32_t     playback;
    int32_t     capture;
    int32_t     card;
    int32_t     device;
    int32_t     shared_card;
    int32_t     shared_device;
    uint8_t     reserved[128];      /* must be filled with
zeroes */
}       snd_pcm_info_t;
```

Description:

The snd_pcm_info_t structure describes the capabilities of a PCM device. The members include:

type

Sound card type. Deprecated. Do not use.

flags

Any combination of:

- SND_PCM_INFO_PLAYBACK — the playback channel is present.
- SND_PCM_INFO_CAPTURE — the capture channel is present.
- SND_PCM_INFO_DUPLEX — the hardware is capable of duplex operation.
- SND_PCM_INFO_DUPLEX_RATE — the playback and capture rates must be same for the duplex operation.
- SND_PCM_INFO_DUPLEX_MONO — the playback and capture must be monophonic for the duplex operation.
- SND_PCM_INFO_SHARED — some or all of the hardware channels are shared using software PCM mixing.

id[64]

ID of this PCM device (user selectable).

name[80]

Name of the device.

playback

Number of playback subdevices - 1.

capture

Number of capture subdevices - 1.

card

Card number.

device

Device number.

shared_card

Number of shared cards for this PCM device.

shared_device

Number of shared devices for this PCM device.

Classification:

QNX Neutrino

snd_pcm_link()

Link two PCM streams together

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_link( snd_pcm_t *pcm1,
                  snd_pcm_t *pcm2 );
```

Arguments:

pcm1*, *pcm2

The handles for the PCM devices, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_link()* function links two PCM streams together such that they always play at the same time. Starting one starts the other, and stopping one stops the other.

Returns:

`EOK` on success, or a positive *errno* value if an error occurred.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_nonblock_mode()

Set or reset the blocking behavior of reads and writes to PCM channels

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_nonblock_mode( snd_pcm_t *handle,
                           int nonblock );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

nonblock

If this argument is nonzero, non-blocking mode is in effect for subsequent calls to *snd_pcm_read()* (p. 277) and *snd_pcm_write()* (p. 283).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_nonblock_mode()* function sets up blocking (default) or nonblocking behavior for a *handle*.

Blocking mode suspends the execution of the client application when there's no room left in the buffer it's writing to, or nothing left to read when reading.

In nonblocking mode, programs aren't suspended, and the read and write functions return immediately with the number of bytes that were read or written by the driver. When used in this way, don't try to use the entire buffer after the call; instead, process the number of bytes returned and call the function again.

Returns:

Zero on success, or a negative error code.

Errors:

-EBADF

Invalid file descriptor. Your *handle* may be corrupt.

-EINVAL

Invalid *handle*.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

If possible, it is recommended that you design your application to call select on the PCM file descriptor, instead of using this function. Asynchronously receiving notification from the driver is much less CPU-intensive than polling it in a non-blocking loop.

snd_pcm_open()

Create a handle and open a connection to a specified audio interface

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_open( snd_pcm_t **handle,
                  int card,
                  int device,
                  int mode );
```

Arguments:

handle

A pointer to a location where *snd_pcm_open()* stores a handle for the audio interface. You'll need this handle when you call the other *snd_pcm_** functions.

card

The card number.

device

The audio device number.

mode

One of:

- `SND_PCM_OPEN_PLAYBACK` — open the playback channel (direction).
- `SND_PCM_OPEN_CAPTURE` — open the capture channel (direction).

You can OR this flag with any of the above:

- `SND_PCM_OPEN_NONBLOCK` — force the mode to be nonblocking. This affects any reading from or writing to the device that you do later; you can query the device any time without blocking.

  You can change the blocking setup later by calling *snd_pcm_nonblock_mode()* (p. 219)

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_open()* function creates a handle and opens a connection to the audio interface for sound card number *card* and audio device number *device*. It also checks if the protocol is compatible to prevent the use of programs written to an older API with newer drivers.

There are no defaults; your application must specify all the arguments to this function.

Using names for audio devices (*snd_pcm_open_name()* (p. 223)) is preferred to using numbers (*snd_pcm_open()*), although *snd_pcm_open_preferred()* (p. 226). remains a good alternative to both.

Returns:

Zero on success, or a negative error code.

Errors:

-ENOMEM

Not enough memory to allocate control structures.

Examples:

See the example in "*Opening your PCM device* (p. 26)" in the Playing and Capturing Audio Data chapter.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Successfully opening a PCM channel doesn't guarantee that there are enough audio resources free to handle your application. Audio resources (e.g., subchannels) are allocated when you configure the channel by calling *snd_pcm_channel_params()* (p. 165) or *snd_pcm_plugin_params()* (p. 247).

snd_pcm_open_name()

Create a handle and open a connection to an audio interface specified by name

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_open_name( snd_pcm_t **handle,
                       char *name,
                       int mode );
```

Arguments:

handle

A pointer to a location where *snd_pcm_open_name()* can store a handle for the audio interface. You'll need this handle when you call the other *snd_pcm_** functions.

name

The name of the PCM device to open; one of the following:

Use this device:	For:
`pcmPreferred`	The default to use under most circumstances
`tones`	The default to use for system sounds. Sounds played through this device aren't attenuated by the global volume setting.
`bluetooth`	Accessing a Bluetooth headset
`voice`	Acoustic echo cancellation and noise suppression, as well as lower latency
`hdmi_mix`	Playback through a receiver connected over HDMI
`usb_mix`	Playback through a receiver connected over USB
`usb`	Capture over a microphone connected over USB

mode

One of:

- `SND_PCM_OPEN_PLAYBACK` — open the playback channel (direction).
- `SND_PCM_OPEN_CAPTURE` — open the capture channel (direction).

You can OR the following flag with any of the above:

- `SND_PCM_OPEN_NONBLOCK` — force the mode to be nonblocking. This affects any reading from or writing to the device that you do later; you can query the device any time without blocking.

  You can change the blocking setup later by calling *snd_pcm_nonblock_mode()* (p. 219).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_open_name()* function creates a handle and opens a connection to the named PCM audio interface.

Using names for audio devices (*snd_pcm_open_name()*) is preferred to using numbers (*snd_pcm_open()* (p. 221)), although *snd_pcm_open_preferred()* (p. 226). remains a good alternative to both.

Returns:

EOK

Success.

-EINVAL

The mode is invalid.

-ENOENT

The named device doesn't exist.

-ENOMEM

Not enough memory is available to allocate the control structures.

-SND_ERROR_INCOMPATIBLE_VERSION

The audio driver version is incompatible with the client library that the application is using.

Examples:

```
snd_pcm_open_name(&pcm_handle, "voice", SND_PCM_OPEN_CAPTURE);
```

See also the example of *snd_pcm_open()* in "*Opening your PCM device* (p. 26)" in the Playing and Capturing Audio Data chapter.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Successfully opening a PCM channel doesn't guarantee that there are enough audio resources free to handle your application. Audio resources (e.g., subchannels) are allocated when you configure the channel by calling *snd_pcm_channel_params()* (p. 165) or *snd_pcm_plugin_params()* (p. 247).

snd_pcm_open_preferred()

Create a handle and open a connection to the preferred audio interface

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_open_preferred( snd_pcm_t **handle,
                            int *rcard,
                            int *rdevice,
                            int mode );
```

Arguments:

handle

A pointer to a location where *snd_pcm_open_preferred()* can store a handle for the audio interface. You'll need this handle when you call the other *snd_pcm_** functions.

rcard

If non-`NULL`, this must be a pointer to a location where *snd_pcm_open_preferred()* can store the number of the card that it opened.

rdevice

If non-`NULL`, this must be a pointer to a location where *snd_pcm_open_preferred()* can store the number of the audio device that it opened.

mode

One of:

- `SND_PCM_OPEN_PLAYBACK` — open the playback channel (direction).
- `SND_PCM_OPEN_CAPTURE` — open the capture channel (direction).

You can OR this flag with any of the above:

- `SND_PCM_OPEN_NONBLOCK` — force the mode to be nonblocking. This affects any reading from or writing to the device that you do later; you can query the device any time without blocking.

  You can change the blocking setup later by calling *snd_pcm_nonblock_mode()* (p. 219).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_open_preferred()* function is an extension to the *snd_pcm_open()* (p. 221) function that attempts to open the user-selected default (or preferred) device for the system.

> If you use this function, your application will be more flexible than if you use *snd_pcm_open()*.

In a system where more than one PCM device exists, the user may set a preference for one of these devices. This function attempts to open that device and return a PCM handle to it. The function returns the card and device numbers if the *rcard* and *rdevice* arguments aren't `NULL`.

Here's the search order to find the preferred device:

1. Read `/etc/system/config/audio/preferences`. The format of this file is as follows, with a tab character between the fields:

```
pcmPreferredp    card_number    device_number
pcmPreferredc    card_number    device_number
```

2. If this file doesn't exist or has no entry, check PCM device 0 of card 0 for a software mixing overlay device. If this overlay device is found, it's opened.
3. Open the default device 0 of card 0.

If all of the above fail, you don't have an audio system running.

Returns:

Zero on success, or a negative value on error.

Errors:

-EINVAL

Invalid *mode*.

-EACCES

Search permission is denied on a component of the path prefix, or the device exists and the permissions specified are denied.

-EINTR

The *open()* operation was interrupted by a signal.

-EMFILE

Too many file descriptors are currently in use by this process.

-ENFILE

Too many files are currently open in the system.

-ENOENT

The named device doesn't exist.

-SND_ERROR_INCOMPATIBLE_VERSION

The audio driver version is incompatible with the client library that the application uses.

-ENOMEM

No memory available for data structures.

Examples:

See the example in "*Opening your PCM device* (p. 26)" in the Playing and Capturing Audio Data chapter.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Successfully opening a PCM channel doesn't guarantee that there are enough audio resources free to handle your application. Audio resources (e.g., subchannels) are allocated when you configure the channel by calling *snd_pcm_channel_params()* (p. 165) or *snd_pcm_plugin_params()* (p. 247).

snd_pcm_playback_drain()

Stop the PCM playback channel and discard the contents of its queue

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_playback_drain( snd_pcm_t *handle );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_playback_drain()* function stops the PCM playback channel associated with *handle* and causes it to discard all audio data in its buffers. This all happens immediately.

If the operation is successful (zero is returned), the channel's state is changed to `SND_PCM_STATUS_READY`.

Returns:

Zero on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle*, or the PCM device state isn't ready.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_playback_flush()

Play out all pending data in a PCM playback channel's queue and stop the channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_playback_flush( snd_pcm_t *handle);
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_playback_flush()* function blocks until all unprocessed data in the driver queue has been played.

If the operation is successful (zero is returned), the channel's state is changed to `SND_PCM_STATUS_READY` and the channel is stopped.

This function *isn't* plugin-aware. It functions exactly the same way as `snd_pcm_channel_flush(.., SND_PCM_CHANNEL_PLAYBACK)`. Make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

Returns:

Zero on success, or a negative error code.

Errors:

-EBADFD

The PCM device state isn't ready.

-EINTR

The driver isn't processing the data (Internal Error).

-EINVAL

Invalid *handle*.

-EIO

An invalid channel was specified, or the data wasn't all flushed.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_playback_go()

Start a PCM playback channel running

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_playback_go ( snd_pcm_t *handle );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Description:

The *snd_pcm_playback_go()* function starts the playback channel running.

You should call this function only when the channel is in the `SND_PCM_STATUS_READY` state, and you should ensure that two or more audio fragments have been written into the audio interface before issuing the go command, to prevent the audio channel/stream from going into the UNDERRUN state.

Calling this function is required if you've set your channel's start state to `SND_PCM_START_GO` (see *snd_pcm_plugin_params()* (p. 247)). You can use this function to “kick start” early a playback channel that has a start state of `SND_PCM_START_DATA` or `SND_PCM_START_FULL`.

If the parameters are valid (i.e, the function returns zero), then the driver state is changed to `SND_PCM_STATUS_RUNNING`.

This function is safe to use with plugin-aware functions. This call is used identically to *snd_pcm_plugin_params()* (p. 247).

Returns:

EOK

Success.

-EINVAL

Invalid *handle*.

-EIO

Invalid *channel*.

-EMORE

Insufficient audio fragments have been written to the audio interface (when writing to the software mixer device).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_playback_pause()

Pause a channel that's playing back

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_playback_pause ( snd_pcm_t *pcm );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_playback_pause()* function pauses a channel that's playing back. Unlike draining or flushing, this preserves all data that has not yet played out within the audio driver, to be played out after resuming.

Returns:

EOK

Success.

-EINVAL

The handle is `NULL`, or the channel isn't playing back.

This function can return other negative *errno* values.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_playback_prepare()

Signal the driver to ready the playback channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_playback_prepare( snd_pcm_t *handle);
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_playback_prepare()* function prepares hardware to operate in a specified transfer direction. This call is responsible for all parts of the hardware's startup sequence that require additional initialization time, allowing the final “GO” (either from writes into the buffers or *snd_pcm_channel_go()* (p. 157)) to execute more quickly.

You can call this function in all states except `SND_PCM_STATUS_NOTREADY` (returns `-EBADFD`) and `SND_PCM_STATUS_RUNNING` (returns `-EBUSY`). If the operation is successful (zero is returned), the driver state is changed to `SND_PCM_STATUS_PREPARED`.

If your channel has underrun, you have to reprepare it before continuing. For an example, see ***`wave.c`*** in the appendix.

Returns:

Zero on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle*.

-EBUSY

Channel is already running.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_playback_resume()

Resume a channel that was paused while playing back

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_playback_resume ( snd_pcm_t *pcm );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_playback_resume()* function resumes a channel that was paused while playing back.

Returns:

EOK

Success.

-EINVAL

The handle is `NULL`, or the channel wasn't being used for playback.

This function can return other negative *errno* values.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_plugin_flush()

Finish processing all pending data in a PCM channel's queue and stop the channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_flush( snd_pcm_t *handle,
                          int channel );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel; `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_flush()* function flushes all unprocessed data in the driver queue:

- If the plugin is processing playback data, the call blocks until all data in the driver queue is played out the channel.
- If the plugin is processing capture data, any unread data in the driver queue is discarded.

If the operation is successful (zero is returned), the channel's state is changed to `SND_PCM_STATUS_READY`.

Returns:

A positive number on success, or a negative value on error.

Errors:

-EINVAL

Invalid *handle*.

Examples:

See the *`wave.c` example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Because the plugin interface may be subbuffering the written data until a complete driver block can be assembled, the flush call may have to inject up to (*blocksize*-1) samples into the channel so that the last block can be sent to the driver for playing. For this reason, the flush call may return a positive value indicating that this silence had to be inserted.

This function is the plugin-aware version of *snd_pcm_channel_flush()* (p. 155). It functions exactly the same way, with the above caveat. However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_get_voice_conversion()

Get the current voice conversion structure for a channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_get_voice_conversion (
        snd_pcm_t *handle,
        int channel,
        snd_pcm_voice_conversion_t *voice_conversion );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel direction; either `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

voice_conversion

A pointer to a `snd_pcm_voice_conversion_t` (p. 282) structure that the function fills in.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_get_voice_conversion()* function gets the current voice conversion structure for the specified channel.

Returns:

`EOK`

Success.

`-EINVAL`

One or more of the arguments were invalid.

-ENOENT

The voice converter plugin doesn't exist.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_plugin_info()

Get information about a PCM channel's capabilities (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_info(
         snd_pcm_t *handle,
         snd_pcm_channel_info_t *info );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

info

A pointer to a `snd_pcm_channel_info_t` (p. 161) structure that *snd_pcm_plugin_info()* fills in with information about the PCM channel.

Before calling this function, set the *info* structure's *channel* member to specify the direction. This function sets all the other members.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_info()* function fills the *info* structure with data about the PCM channel selected by *handle*.

This function and the nonplugin version, *snd_pcm_channel_info()* (p. 159), get a dynamic "snapshot" of the system's current capabilities, which can shrink and grow as subchannels are allocated and freed. They're similar to *snd_ctl_pcm_channel_info()* (p. 81), which gets information about the *complete* capabilities of the system.

Returns:

Zero on success, or a negative error code.

Errors:

-EINVAL

Invalid *handle*.

Examples:

See the *wave.c example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is the plugin-aware version of *snd_pcm_channel_info()* (p. 159). It functions exactly the same way. However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_params()

Set the configurable parameters for a PCM channel (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_params(
        snd_pcm_t *handle,
        snd_pcm_channel_params_t *params );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

params

A pointer to a `snd_pcm_channel_params_t` (p. 167) structure in which you've specified the PCM channel's configurable parameters. All members are write-only.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_params()* function sets up the transfer parameters according to *params*.

You can call the function in `SND_PCM_STATUS_NOTREADY` (initial) and `SND_PCM_STATUS_READY` states; otherwise, *snd_pcm_plugin_params()* returns `-EBADFD`.

If the parameters are valid (i.e., *snd_pcm_plugin_params()* returns zero), the driver state is changed to `SND_PCM_STATUS_READY`.

You can confirm the channel's configuration by reading it back with *snd_pcm_plugin_setup()* (p. 266).

Returns:

Zero, or a negative error code.

Errors:

-EINVAL

Invalid *handle*; the data pointer is `NULL`, or the format is unsupported.

Examples:

See the *`wave.c` example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is the plugin-aware version of *snd_pcm_channel_params()* (p. 165). It functions exactly the same way. However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_playback_drain()

Stop the PCM playback channel and discard the contents of its queue (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_playback_drain(
        snd_pcm_t *handle );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_playback_drain()* function stops the PCM playback channel associated with *handle* and causes it to discard all audio data in its buffers. This happens immediately.

If the operation is successful (zero is returned), the channel's state is changed to `SND_PCM_STATUS_READY`.

Returns:

Zero on success, or a negative error code (*errno* is set).

Errors:

-EINVAL

Invalid *handle*, or the PCM device state isn't ready.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is the plugin-aware version of *snd_pcm_playback_drain()* (p. 229). It functions exactly the same way. However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_prepare()

Signal the driver to ready the specified channel (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_prepare( snd_pcm_t *handle,
                            int channel );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel; `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_prepare()* function prepares hardware to operate in a specified transfer direction. This call is responsible for all parts of the hardware's startup sequence that require additional initialization time, allowing the final "GO" (either from writes into the buffers or *snd_pcm_channel_go()* (p. 157)) to execute more quickly.

This function may be called in all states except `SND_PCM_STATUS_NOTREADY` (returns `-EBADFD`) and `SND_PCM_STATUS_RUNNING` (returns `-EBUSY`). If the operation is successful (zero is returned), the driver state is changed to `SND_PCM_STATUS_PREPARED`.

If your channel has underrun (during playback) or overrun (during capture), you have to reprepare it before continuing. For an example, see `wave.c` and `waverec.c` in the appendix.

Returns:

Zero, or a negative error code.

Errors:

-EBUSY

The subchannel is in the running state.

-EINVAL

Invalid *handle*.

Examples:

See the *`wave.c` example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is the plugin-aware version of *snd_pcm_channel_prepare()* (p. 173). It functions exactly the same way. However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_read()

Transfer PCM data from the capture channel (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

ssize_t snd_pcm_plugin_read( snd_pcm_t *handle,
                             void *buffer,
                             size_t size );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

buffer

A pointer to a buffer in which *snd_pcm_plugin_read()* can store the data that it reads.

size

The size of the buffer, in bytes.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_read()* function reads samples from the device which must be in the proper format specified by *snd_pcm_plugin_params()* (p. 247).

The *handle* and the *buffer* must be valid.

This function may suspend the client application if block behavior is active (see *snd_pcm_nonblock_mode()* (p. 219)) and no data is available for reading.

Returns:

A positive value that represents the number of bytes that were successfully read from the device if the capture was successful, or a negative value if an error occurred.

Errors:

-EFAULT

Failed to copy data.

-EINVAL

Partial block buffering is disabled, but the *size* isn't the full block size.

-EIO

The channel isn't in the prepared or running state.

-ENOMEM

Unable to allocate memory for plugin buffers.

If you're reading less than a fragment-sized block, you won't get an `-EFAULT` or `-EIO` error until enough read operations have been completed to read the fragment size. The sub-buffering plugin buffers all operations until there is a fragment's worth of data, at which point the message to `io-audio` occurs (you can't get an error until the request goes to `io-audio`).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is the plugin-aware version of *snd_pcm_read()* (p. 277). It functions exactly the same way, with only one caveat (see below). However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

The plugin-aware versions of the PCM read and write calls don't require that you work with multiples of fragment-size blocks (the nonplugin-aware versions do). This is because one of the plugins in the lib sub-buffers the data for you. You can disable this plugin by setting the `PLUGIN_DISABLE_BUFFER_PARTIAL_BLOCKS` bit with *snd_pcm_plugin_set_disable()* (p. 256), in which case, the plugin-aware versions also fail on reads and writes that aren't multiples of the fragment size.

Either way, interleaved stereo data has to be aligned by the sample size times the number of channels (i.e., each write must have the same number of samples for the left and right channels).

snd_pcm_plugin_set_disable()

Disable PCM plugins

Synopsis:

```
#include <sys/asoundlib.h>

unsigned int snd_pcm_plugin_set_disable(
                            snd_pcm_t *pcm,
                            unsigned int mask );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

mask

Currently, only the following *mask* bits are supported:

- `PLUGIN_DISABLE_MMAP` — disable the mmap plugins.

  If mmap plugins are used, some of the members of the `snd_pcm_channel_status_t` (p. 184) structure aren't used.

- `PLUGIN_DISABLE_BUFFER_PARTIAL_BLOCKS` — prevent the read and write routines from using partial blocks of data.

  The plugin-aware versions of the PCM read and write calls don't require that you work with multiples of fragment-size blocks (the nonplugin-aware versions do). This is because one of the plugins in the lib sub-buffers the data for you. You can disable this plugin by setting the `PLUGIN_DISABLE_BUFFER_PARTIAL_BLOCKS` bit with this function, in which case the plugin-aware versions also fail on reads and writes that aren't multiples of the fragment size.

  Either way, interleaved stereo data has to be aligned by the sample size times the number of channels (i.e., each write must have the same number of samples for the left and right channels).

- `PLUGIN_CONVERSION` — disable the automatic conversion of audio to match hardware capabilities (for example, voice conversion, rate conversion, type conversion, etc.). This conversion impacts the functions *snd_pcm_channel_params()* (p. 165), *snd_pcm_channel_setup()* (p. 177), and *snd_pcm_channel_status()* (p. 182). These now behave as *snd_pcm_plugin_params()* (p. 247), *snd_pcm_plugin_setup()* (p. 266), and *snd_pcm_plugin_status()* (p. 270), unless you've disabled the conversion by calling:

```
    snd_pcm_plugin_set_disable(handle,
PLUGIN_CONVERSION);
```

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_set_disable()* function is used to disable various plugins that would ordinarily be used in the plugin chain.

Returns:

The value of the plugin *mask* before this change was made.

Examples:

See the *`wave.c` example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

You need to set the plugin disable mask before calling *snd_pcm_plugin_params()* for it to take effect.

snd_pcm_plugin_set_enable()

Enable plugins that have been disabled

Synopsis:

```
#include <sys/asoundlib.h>

unsigned int snd_pcm_plugin_set_enable( snd_pcm_t *pcm,
                                        unsigned int mask );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

mask

A bitset of the sets you want to disable.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_set_enable()* function enables plugins that have been disabled. Currently there is one disabled plugin: `PLUGIN_ROUTING`. Enabling this will cause automatic routing to external devices for a client that connects to the `"pcmPreferred"` device.

Returns:

The value of the plugin *mask* before this change was made.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_plugin_set_src_method()

Set the system's source filter method (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

unsigned int snd_pcm_plugin_set_src_method (
                snd_pcm_t *handle,
                unsigned int method );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

method

The filter that you want to use:

- 0 — basic linear interpolated SRC (the default)
- 1 — basic anti-aliased SRC filter (7-point Kaiser windowed)

Library:

`libasound.so`

Description:

The *snd_pcm_plugin_set_src_method()* function sets the source filter method. If you want to set this method, do so before you call *snd_pcm_plugin_params()* (p. 247), so that the plugin can be properly initialized (including the filters).

Returns:

The current method.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_set_src_mode()

Set the system's source mode (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

unsigned int snd_pcm_plugin_set_src_mode(
                snd_pcm_t *handle,
                unsigned int src_mode,
                int target );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

src_mode

The sample rate conversion mode; one of the following:

- `SND_SRC_MODE_NORMAL` — (default mode; all previous version of SRC work this way) SRC ratio based on input/output block size rounded towards zero. Floor(input size/output size).
- `SND_SRC_MODE_ACTUAL` — fixed SRC which adjusts the input fragment size dynamically to prevent roundoff error from adjusting the playback speed.
- `SND_SRC_MODE_ASYNC` — asynchronous SRC which adjusts the input fragment size to maintain a specified buffer fullness.

target

The level in percent for the buffer fullness measurement used in the asynchronous sample rate conversion.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_set_src_mode()* function sets the type of sample rate conversion to use. Only playback modes are supported.

Returns:

The source mode (also in *handle->plugin_src_mode*) that the system is set to.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_set_voice_conversion()

Set the current voice conversion structure for a channel

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_set_voice_conversion (
        snd_pcm_t *handle,
        int channel,
        snd_pcm_voice_conversion_t *voice_conversion );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

channel

The channel direction; either `SND_PCM_CHANNEL_CAPTURE` or `SND_PCM_CHANNEL_PLAYBACK`.

voice_conversion

A pointer to a `snd_pcm_voice_conversion_t` (p. 282) structure that specifies how to convert the voices.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_set_voice_conversion()* function sets the current voice conversion structure for the specified channel.

Returns:

`EOK`

Success.

`-EINVAL`

One or more of the arguments were invalid.

-ENOENT

The voice converter plugin doesn't exist.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_plugin_setup()

Get the current configuration for the specified PCM channel (plugin aware)

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_setup(
        snd_pcm_t *handle,
        snd_pcm_channel_setup_t *setup );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

setup

A pointer to a `snd_pcm_channel_setup_t` (p. 179) structure that *snd_pcm_plugin_setup()* fills with information about the current configuration of the PCM channel.

Set the *setup* structure's *channel* member to specify the direction. All other members are read-only.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_setup()* function fills the *setup* structure with information about the current configuration of the PCM channel selected by *handle*.

Returns:

Zero on success, or a negative error code.

Errors:

`-EINVAL`

Invalid *handle*; data pointer is `NULL`; *setup->mode* isn't `SND_PCM_MODE_BLOCK`.

Examples:

See the *`wave.c` example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is the plugin-aware version of *snd_pcm_channel_setup()* (p. 177). It functions exactly the same way. However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_src_max_frag()

Get the maximum possible fragment size (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_src_max_frag ( snd_pcm_t *handle,
                                   unsigned int fragsize );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

fragsize

The fragment size.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_src_max_frag()* function returns the maximum possible fragment size when the system is using the `SND_SRC_MODE_ACTUAL` or `SND_SRC_MODE_ASYNC` mode. The fragment size is adjusted during playback, so this lets you preallocate the maximum buffer size.

Returns:

The maximum fragment size, or `-EINVAL` if any of the arguments were invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_status()

Get the runtime status of a PCM channel (plugin aware)

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_status(
        snd_pcm_t *handle,
        snd_pcm_channel_status_t *status );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

status

A pointer to a `snd_pcm_channel_status_t` (p. 184) structure that *snd_pcm_plugin_status()* fills with information about the PCM channel's status.

Fill in the *status* structure's *channel* member to specify the direction. All other members are read-only.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_status()* function fills the *status* structure with runtime status information about the PCM channel selected by *handle*.

Returns:

Zero on success, or a negative error code.

Errors:

-EBADFD

The PCM device state isn't ready.

-EFAULT

Failed to copy data.

-EINVAL

Invalid *handle* or the data pointer is `NULL`.

Examples:

See the *`wave.c` example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is the plugin-aware version of *snd_pcm_channel_status()* (p. 182). It functions exactly the same way. However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_update_src()

Get the size of the next fragment to write (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_plugin_update_src(
        snd_pcm_t *handle,
        snd_pcm_channel_setup_t *setup,
        int currlevel );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

setup

A pointer to a `snd_pcm_channel_setup_t` (p. 179) structure that *snd_pcm_plugin_setup()* fills with information about the current configuration of the PCM channel.

currlevel

The current level of client size buffering, in percent.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_update_src()* function returns the size of the next fragment required for *snd_pcm_plugin_write()* (p. 274).

If you're using SND_SRC_MODE_ACTUAL or SND_SRC_MODE_ASYNC mode (see *snd_pcm_plugin_set_src_mode()* (p. 262)), you need to call *snd_pcm_plugin_update_src()* after each call to *snd_pcm_plugin_write()*.

The client is responsible for buffering an appropriate amount of data in order to not underflow the write calls. The client must determine the buffer fullness in percent (number of PCM samples the client is holding divided by the total buffer space available). The sample rate converter in `libasound` adjusts the sample rate converter to maintain a close tracking of the target (in percent) set in *snd_pcm_plugin_update_src()*.

Returns:

The number of samples to write in the next *snd_pcm_plugin_write()* call, or `-EINVAL` if any of the arguments are invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

Make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

snd_pcm_plugin_write()

Transfer PCM data to playback channel (plugin-aware)

Synopsis:

```
#include <sys/asoundlib.h>

ssize_t snd_pcm_plugin_write( snd_pcm_t *handle,
                              const void *buffer,
                              size_t size );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

buffer

A pointer to a buffer that contains the data to be written.

size

The size of the data, in bytes.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_plugin_write()* function writes samples that are in the proper format specified by *snd_pcm_plugin_params()* (p. 247) to the device specified by *handle*.

The *handle* and the *buffer* must be valid.

If you're using SND_SRC_MODE_ACTUAL or SND_SRC_MODE_ASYNC mode (see *snd_pcm_plugin_set_src_mode()* (p. 262)), you need to call *snd_pcm_plugin_update_src()* (p. 272) after each call to *snd_pcm_plugin_write()*.

Returns:

A positive value that represents the number of bytes that were successfully written to the device if the playback was successful. A value less than the write request size is an indication of an error; for more information, check the *errno* value and call *snd_pcm_plugin_status()* (p. 270).

Errors:

EAGAIN

Try again later. The subchannel is opened nonblock.

EINVAL

Partial block buffering is disabled, but the *size* isn't the full block size.

EIO

One of:

- The channel isn't in the prepared or running state.

  If you're writing less than a fragment-sized block, you won't get this error until enough write operations have been completed to write the fragment size. The sub-buffering plugin buffers all operations until there's a fragment's worth of data, at which point the message to `io-audio` occurs (you can't get an error until the request goes to `io-audio`).

- In `SND_PCM_MODE_BLOCK` mode, the *size* isn't an even multiple of the *frag_size* member of the `snd_pcm_channel_setup_t` (p. 179) structure and PCM subbuffering has been disabled with *snd_pcm_plugin_set_disable()* (p. 256).

EWOULDBLOCK

The write would have blocked (nonblocking write).

Examples:

See the *`wave.c` example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is the plugin-aware version of *snd_pcm_write()* (p. 283). It functions exactly the same way, with one caveat (see below). However, make sure that you don't mix and match plugin- and nonplugin-aware functions in your application, or you may get undefined behavior and misleading results.

The plugin-aware versions of the PCM read and write calls don't require that you work with multiples of fragment-size blocks (the nonplugin-aware versions do). This is because one of the plugins in the lib sub-buffers the data for you. You can disable this plugin by setting the `PLUGIN_DISABLE_BUFFER_PARTIAL_BLOCKS` bit with *snd_pcm_plugin_set_disable()* (p. 256), in which case, the plugin-aware versions also fail on reads and writes that aren't multiples of the fragment size.

Either way, interleaved stereo data has to be aligned by the sample size times the number of channels (i.e., each write must have the same number of samples for the left and right channels).

snd_pcm_read()

Transfer PCM data from the capture channel

Synopsis:

```
#include <sys/asoundlib.h>

ssize_t snd_pcm_read( snd_pcm_t *handle,
                      void *buffer,
                      size_t size );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

buffer

A pointer to a buffer in which *snd_pcm_read()* can store the data that it reads.

size

The size of the buffer, in bytes.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_read()* function reads samples from the device, which must be in the proper format specified by *snd_pcm_channel_prepare()* (p. 173) or *snd_pcm_capture_prepare()* (p. 151).

This function may suspend the client application if the blocking behavior is active (see *snd_pcm_nonblock_mode()* (p. 219)) and no data is available to be read.

When the subdevice is in block mode (`SND_PCM_MODE_BLOCK`), then the number of read bytes must fulfill the $N \times$ fragment-size expression, where $N > 0$.

If the stream format is noninterleaved (i.e., the *interleave* member of the `snd_pcm_format_t` (p. 204) structure isn't set), then the driver returns data that's separated to single voice blocks encapsulated to fragments. For example, imagine you have two voices, and the fragment size is 512 bytes. The number of bytes per one voice is 256. The driver returns the first 256 bytes that contain samples for the first voice, and the second 256 bytes from the fragment size that contains samples for the second voice.

Returns:

A positive value that represents the number of bytes that were successfully read from the device if the capture was successful, or a negative value if an error occurred.

Errors:

`-EAGAIN`

The subdevice has no data available.

`-EFAULT`

Failed to copy data.

`-EINVAL`

Invalid *handle*; data pointer is `NULL` but size isn't zero or is negative.

`-EIO`

The channel isn't in the prepared or running state.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_set_audioman_handle()

Bind an audioman handle to a PCM stream

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_set_audioman_handle( snd_pcm_t *handle,
                                 unsigned int audioman_handle
 );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

audioman_handle

The handle for the audio manager that you want to bind to the PCM stream.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_set_audioman_handle()* function binds an audioman handle to a PCM stream. Binding an audioman handle to a PCM stream results in the PCM stream's conditionally ducking behind other streams, depending on the type of other stream playing. Rebinding a PCM stream to a new handle is permitted; doing so automatically unbinds the old handle. Binding the same audioman handle to two PCM streams isn't permitted.

Returns:

EOK

Success.

-EINVAL

The PCM handle is `NULL`, or the audioman handle couldn't be bound to the stream.

-EPERM

The thread doesn't have permission to bind the audioman handle to the stream

This function can also return the negative of other *errno* values.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_unlink()

Detach a PCM stream from a link group

Synopsis:

```
#include <sys/asoundlib.h>

int snd_pcm_unlink( snd_pcm_t *pcm );
```

Arguments:

pcm

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_unlink()* function detaches a PCM stream from a link group. After this point, starting and stopping this PCM stream affects the stream only, not any other streams.

Returns:

0 on success, or -1 if an error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_pcm_voice_conversion_t

Data structure that controls voice conversion

Synopsis:

```
typedef struct snd_pcm_voice_conversion
{
    uint32_t    app_voices;
    uint32_t    hw_voices;
    uint32_t    matrix[32];
}       snd_pcm_voice_conversion_t;
```

Description:

The `snd_pcm_voice_conversion_t` structure controls how the voice-converter plugin replicates or reduces the voices and channels.

The members include:

app_voices

The number of application voices.

hw_voices

The number of hardware voices.

matrix

A 32-by-32-bit array that specifies how to convert the voices. The array is ranked with rows representing application voices, voice 0 first; the columns represent hardware voices, with the low voice being LSB-aligned and increasing right to left. A 1 in an entry directs the given source to the given destination.

Classification:

QNX Neutrino

snd_pcm_write()

Transfer PCM data to playback channel

Synopsis:

```
#include <sys/asoundlib.h>

ssize_t snd_pcm_write( snd_pcm_t *handle,
                       const void *buffer,
                       size_t size );
```

Arguments:

handle

The handle for the PCM device, which you must have opened by calling *snd_pcm_open_name()* (p. 223), *snd_pcm_open()* (p. 221), or *snd_pcm_open_preferred()* (p. 226).

buffer

A pointer to a buffer that holds the data to be written.

size

The amount of data to write, in bytes.

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_pcm_write()* function writes samples to the device, which must be in the proper format specified by *snd_pcm_channel_prepare()* (p. 173) or *snd_pcm_playback_prepare()* (p. 237).

This function may suspend a process if blocking behavior is active (see *snd_pcm_nonblock_mode()* (p. 219)) and no space is available in the device's buffers.

When the subdevice is in block mode (`SND_PCM_MODE_BLOCK`), then the number of written bytes must fulfill the $N \times$ fragment-size expression, where $N > 0$.

If the stream format is noninterleaved (the *interleave* member of the `snd_pcm_format_t` (p. 204) structure isn't set), then the driver expects that data in one fragment is separated to single voice blocks. For example, imagine that you have two voices, and the fragment size is 512 bytes. The number of bytes per one voice is 256. The driver expects that the first 256 bytes contain samples for the first voice and the second 256 bytes from fragment contain samples for the second voice.

Returns:

A positive value that represents the number of bytes that were successfully written to the device if the playback was successful, or an error value if an error occurred.

Errors:

`-EAGAIN`

Try again later. The subchannel is opened nonblock.

`-EINVAL`

One of the following:

- The handle is invalid.
- The *buffer* argument is `NULL`, but the *size* is greater than zero.
- The *size* is negative.

`-EIO`

One of:

- The channel isn't in the prepared or running state.
- In `SND_PCM_MODE_BLOCK` mode, the *size* isn't an even multiple of the *frag_size* member of the `snd_pcm_channel_setup_t` (p. 179) structure.

`-EWOULDBLOCK`

The write would have blocked (nonblocking write).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_strerror()

Convert an error code to a string

Synopsis:

```
#include <sys/asoundlib.h>

const char *snd_strerror( int errnum );
```

Arguments:

errnum

An error number, which can be positive (i.e., the value of *errno*) or negative (i.e., a return code from a *snd_** function).

Library:

`libasound.so`

Use the `-l asound` option to `qcc` to link against this library.

Description:

The *snd_strerror()* function converts an error code to a string. Its functionality is similar to that of *strerror()* (see the QNX Neutrino *C Library Reference*), except that it returns the correct strings for sound error codes.

Returns:

A pointer to the error message. Don't modify the string that it points to.

If *snd_strerror()* doesn't recognize the value for *errnum*, it returns a pointer to the string "`Unknown error`."

Examples:

See the *`wave.c` example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

snd_switch_t

Information about a mixer's switch

Synopsis:

```
typedef struct snd_switch
{
    int32_t iface;
    int32_t device;
    int32_t channel;
    char    name[36];
    uint32_t type;
    uint32_t subtype;
    uint32_t zero[2];
    union
    {
        uint32_t enable:1;

        struct
        {
            uint8_t data;
            uint8_t low;
            uint8_t high;
        }
        byte;

        struct
        {
            uint16_t data;
            uint16_t low;
            uint16_t high;
        }
        word;

        struct
        {
            uint32_t data;
            uint32_t low;
            uint32_t high;
        }
        dword;

        struct
        {
            uint32_t data;
            uint32_t items[30];
            uint32_t items_cnt;
        }
        list;

        struct
        {
            uint8_t selection;
            char    strings[11][11];
            uint8_t strings_cnt;
        }
        string_11;
```

```
        uint8_t raw[32];
        uint8_t reserved[128];      /* must be filled with
zeroes */
    }
    value;
    uint8_t reserved[128];      /* must be filled with zeroes
 */
}
snd_switch_t;
```

Description:

The `snd_switch_t` structure describes the switches for a mixer. You can fill this structure by calling *snd_ctl_mixer_switch_read()* (p. 75).

The members include:

iface

The audio interface associated with the switch.

device

The device number associated with the switch.

channel

Currently only set to “0”.

name

The text name of the switch.

type

The kind of switch. The following types are supported:

SND_SW_TYPE_BOOLEAN

A simple on and off switch. See the *enable* union member.

SND_TYPE_BYTE

An 8-bit value constrained between a minimum and maximum setting. See the *byte* union member.

SND_TYPE_WORD

A 16-bit value constrained between a minimum and maximum setting. See the *word* union member.

SND_TYPE_DWORD

A 32-bit value constrained between a minimum and maximum setting. See the *dword* union member.

SND_TYPE_LIST

A 32-bit value selected from a list of values. See the *list* union member. The *items_cnt* argument is the number of valid items in the array.

SND_TYPE_STRING_11

An array of string selections with a maximum length of 11 bytes. The *strings_cnt* argument is the number of valid strings in the array. The *selection* argument is the index of the selected string.

subtype

The switch's subtype. The following types are supported:

SND_SW_SUBTYPE_DEC

Display the value in decimal notation.

SND_SW_SUBTYPE_HEXA

Display the value in hexadecimal notation.

Classification:

QNX Neutrino

Appendix A
wave.c example

This is a sample application that plays back audio data.

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#include <errno.h>
#include <fcntl.h>
#include <gulliver.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/stat.h>
#include <sys/termio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/slogcodes.h>
#include <time.h>
#include <ctype.h>
#include <limits.h>
#include <signal.h>
#include <pthread.h>
#include <sys/pps.h>

#include <sys/asoundlib.h>

#include <audio/audio_manager_volume.h>
#include <audio/audio_manager_routing.h>
#include <audio/audio_manager_event.h>

const char *kRiffId = "RIFF";
const char *kRifxId = "RIFX";
const char *kWaveId = "WAVE";
bool running = true;
int n;
int N=0;
int verbose = 0;
int print_timing = 0;
int bsize;
snd_mixer_group_t group;
snd_mixer_t *mixer_handle = NULL;
snd_pcm_t *pcm_handle;
char   *mSampleBfr1;
unsigned int mSamples;
bool mBigEndian = false;

typedef struct
{
    char    tag[4];
    long    length;
}
RiffTag;

typedef struct
```

```
{
    char    Riff[4];
    long    Size;
    char    Wave[4];
}
RiffHdr;


typedef struct
{
    short   FormatTag;
    short   Channels;
    long    SamplesPerSec;
    long    AvgBytesPerSec;
    short   BlockAlign;
    short   BitsPerSample;
}
WaveHdr;

typedef struct
{
    FILE *file1;
    struct timespec start_time;
}
WriterData;

int
err (char *msg)
{
    perror (msg);
    return -1;
}


int
FindTag (FILE * fp, const char *tag)
{
    int     retVal;
    RiffTag tagBfr = { "", 0 };

    retVal = 0;

    // Keep reading until we find the tag or hit the EOF.
    while (fread ((unsigned char *) &tagBfr, sizeof (tagBfr), 1, fp))
    {

        if( mBigEndian ) {
            tagBfr.length = ENDIAN_BE32 (tagBfr.length);
        } else {
            tagBfr.length = ENDIAN_LE32 (tagBfr.length);
        }
        // If this is our tag, set the length and break.
        if (strncmp (tag, tagBfr.tag, sizeof tagBfr.tag) == 0)
        {
            retVal = tagBfr.length;
            break;
        }

        // Skip ahead the specified number of bytes in the stream
        fseek (fp, tagBfr.length, SEEK_CUR);
    }

    // Return the result of our operation
    return (retVal);
}


int
CheckHdr (FILE * fp)
{
    RiffHdr riffHdr = { "", 0 };

    // Read the header and, if successful, play the file
    // file or WAVE file.
    if (fread ((unsigned char *) &riffHdr, sizeof (RiffHdr), 1, fp) == 0)
        return 0;

    if (!strncmp (riffHdr.Riff, kRiffId, strlen (kRiffId)))
        mBigEndian = false;
    else if (!strncmp (riffHdr.Riff, kRifxId, strlen (kRifxId)))
```

```
        mBigEndian = true;
    else
        return -1;
    if (strncmp (riffHdr.Wave, kWaveId, strlen (kWaveId)))
        return -1;

    return 0;
}


int
dev_raw (int fd)
{
    struct termios termios_p;

    if (tcgetattr (fd, &termios_p))
        return (-1);

    termios_p.c_cc[VMIN] = 1;
    termios_p.c_cc[VTIME] = 0;
    termios_p.c_lflag &= ~(ICANON | ECHO | ISIG);
    return (tcsetattr (fd, TCSANOW, &termios_p));
}

int
dev_unraw (int fd)
{
    struct termios termios_p;

    if (tcgetattr (fd, &termios_p))
        return (-1);

    termios_p.c_lflag |= (ICANON | ECHO | ISIG);
    return (tcsetattr (fd, TCSAFLUSH, &termios_p));
}

void
handle_keypress()
{
    int     c;
    int     rtn;

    c = getc (stdin);

    if (c == EOF)
    {
        running = false;
        return;
    }

    /* Handle non-mixer keypresses */
    switch (c)
    {
        case 'p':
            snd_pcm_playback_pause( pcm_handle );
            break;
        case 'r':
            snd_pcm_playback_resume( pcm_handle );
            break;
  // Exit the program
        case 3: // Ctrl-C
        case 27: // Escape
            running = false;
            break;
        default:
            break;
    }

 /* Handle mixer keypresses */
    if (mixer_handle == NULL)
        return;

    if ((rtn = snd_mixer_group_read (mixer_handle, &group)) < 0)
 {
        fprintf (stderr, "snd_mixer_group_read failed: %s\n", snd_strerror (rtn));
  return;
 }

 switch (c)
 {
```

```
        case 'q':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
                group.volume.names.front_left += 10;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
                group.volume.names.rear_left += 10;
            if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
                group.volume.names.woofer += 10;
            break;
        case 'a':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
                group.volume.names.front_left -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
                group.volume.names.rear_left -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
                group.volume.names.woofer -= 10;
            break;
        case 'w':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
                group.volume.names.front_left += 10;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
                group.volume.names.rear_left += 10;
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
                group.volume.names.front_center += 10;
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
                group.volume.names.front_right += 10;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
                group.volume.names.rear_right += 10;
            if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
                group.volume.names.woofer += 10;
            break;
        case 's':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
                group.volume.names.front_left -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
                group.volume.names.rear_left -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
                group.volume.names.front_center -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
                group.volume.names.front_right -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
                group.volume.names.rear_right -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
                group.volume.names.woofer -= 10;
            break;
        case 'e':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
                group.volume.names.front_right += 10;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
                group.volume.names.rear_right += 10;
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
                group.volume.names.front_center += 10;
            break;
        case 'd':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
                group.volume.names.front_right -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
                group.volume.names.rear_right -= 10;
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
                group.volume.names.front_center -= 10;
            break;
    }

    if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
    {
        if (group.volume.names.front_left > group.max)
            group.volume.names.front_left = group.max;
        if (group.volume.names.front_left < group.min)
            group.volume.names.front_left = group.min;
    }
    if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
    {
        if (group.volume.names.rear_left > group.max)
            group.volume.names.rear_left = group.max;
        if (group.volume.names.rear_left < group.min)
            group.volume.names.rear_left = group.min;
    }
    if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
    {
        if (group.volume.names.front_center > group.max)
            group.volume.names.front_center = group.max;
```

```
        if (group.volume.names.front_center < group.min)
            group.volume.names.front_center = group.min;
    }
    if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
    {
        if (group.volume.names.front_right > group.max)
            group.volume.names.front_right = group.max;
        if (group.volume.names.front_right < group.min)
            group.volume.names.front_right = group.min;
    }
    if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
    {
        if (group.volume.names.rear_right > group.max)
            group.volume.names.rear_right = group.max;
        if (group.volume.names.rear_right < group.min)
            group.volume.names.rear_right = group.min;
    }
    if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
    {
        if (group.volume.names.woofer > group.max)
            group.volume.names.woofer = group.max;
        if (group.volume.names.woofer < group.min)
            group.volume.names.woofer = group.min;
    }
    if ((rtn = snd_mixer_group_write (mixer_handle, &group)) < 0)
        fprintf (stderr, "snd_mixer_group_write failed: %s\n", snd_strerror (rtn));

    if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
    {
        printf ("Volume Now at %d:%d \n",
            (group.max - group.min) ? 100 * (group.volume.names.front_left - group.min) /
               (group.max - group.min) : 0,
            (group.max - group.min) ? 100 * (group.volume.names.front_right - group.min) /
               (group.max - group.min): 0);
    }
    else if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
    {
        printf ("Volume Now at %d:%d \n",
            (group.max - group.min) ? 100 * (group.volume.names.rear_left - group.min) /
               (group.max - group.min) : 0,
            (group.max - group.min) ? 100 * (group.volume.names.rear_right - group.min) /
               (group.max - group.min): 0);
    }
    else if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
    {
        printf ("Volume Now at %d:%d \n",
            (group.max - group.min) ? 100 * (group.volume.names.woofer - group.min) /
               (group.max - group.min) : 0,
            (group.max - group.min) ? 100 * (group.volume.names.front_center - group.min) /
               (group.max - group.min): 0);
    }
    else
    {
        printf ("Volume Now at %d:%d \n",
            (group.max - group.min) ? 100 * (group.volume.names.front_left - group.min) /
               (group.max - group.min) : 0,
            (group.max - group.min) ? 100 * (group.volume.names.front_right - group.min) /
               (group.max - group.min): 0);
    }
}

void handle_mixer()
{
    fd_set  rfds;
    FD_ZERO(&rfds);
    FD_SET (snd_mixer_file_descriptor (mixer_handle), &rfds);

    if (select (snd_mixer_file_descriptor (mixer_handle) + 1, &rfds, NULL, NULL, NULL) == -1)
        err ("select");

    snd_mixer_callbacks_t callbacks = { 0, 0, 0, 0 };

    snd_mixer_read (mixer_handle, &callbacks);
}

void write_audio_data(WriterData *wd)
{
    struct  timespec current_time;
    snd_pcm_channel_status_t status;
    int     written = 0;
```

```
    if ((n = fread (mSampleBfr1, 1, min (mSamples - N, bsize), wd->file1)) <= 0)
        return;
    written = snd_pcm_plugin_write (pcm_handle, mSampleBfr1, n);

    if (verbose)
        printf ("bytes written = %d \n", written);
    if( print_timing ) {
        clock_gettime( CLOCK_REALTIME, &current_time );
        printf ("Sent frag at %llu\n", (current_time.tv_sec - wd->start_time.tv_sec) *
                1000000000LL + (current_time.tv_nsec - wd->start_time.tv_nsec));
    }
    if (written < n)
    {
        memset (&status, 0, sizeof (status));
        status.channel = SND_PCM_CHANNEL_PLAYBACK;
        if (snd_pcm_plugin_status (pcm_handle, &status) < 0)
        {
            fprintf (stderr, "underrun: playback channel status error\n");
            exit (1);
        }

        if (status.status == SND_PCM_STATUS_READY ||
            status.status == SND_PCM_STATUS_UNDERRUN ||
            status.status == SND_PCM_STATUS_CHANGE)
        {
            if( status.status == SND_PCM_STATUS_UNDERRUN ) {
                printf ("Audio underrun occurred\n");
            } else if( status.status == SND_PCM_STATUS_CHANGE ) {
                printf ("Audio device change occurred from %s to %s\n",
                        audio_manager_get_device_name(status.status_data.change_data.old_device),
                        audio_manager_get_device_name(status.status_data.change_data.new_device));
            }
            if (snd_pcm_plugin_prepare (pcm_handle, SND_PCM_CHANNEL_PLAYBACK) < 0)
            {
                fprintf (stderr, "underrun: playback channel prepare error\n");
                exit (1);
            }
        }
        else if (status.status == SND_PCM_STATUS_UNSECURE)
        {
            fprintf (stderr, "Channel unsecure\n");
            if (snd_pcm_plugin_prepare (pcm_handle, SND_PCM_CHANNEL_PLAYBACK) < 0)
            {
                fprintf (stderr, "unsecure: playback channel prepare error\n");
                exit (1);
            }
        }
        else if (status.status == SND_PCM_STATUS_ERROR)
        {
            fprintf(stderr, "error: playback channel failure\n");
            exit(1);
        }
        else if (status.status == SND_PCM_STATUS_PREEMPTED)
        {
            fprintf(stderr, "error: playback channel preempted\n");
            exit(1);
        }

        if (written < 0)
            written = 0;
        written += snd_pcm_plugin_write (pcm_handle, mSampleBfr1 + written, n - written);
    }
    N += written;
}

void *writer_thread_handler(void *data)
{
    WriterData *wd = (WriterData *)data;
    sigset_t signals;

    sigfillset (&signals);
    pthread_sigmask (SIG_BLOCK, &signals, NULL);

    while (running && N < mSamples && n > 0)
    {
        write_audio_data(wd);
    }

    return NULL;
```

```
}

void *generic_thread_handler(void *data)
{
    sigset_t signals;

    sigfillset (&signals);
    pthread_sigmask (SIG_BLOCK, &signals, NULL);

    while(1) {
        ((void (*)(void))data)();
    }

    return NULL;
}


//*****************************************************************************
/* *INDENT-OFF* */
#ifdef __USAGE
%C[Options] *

Options:
    -a[card#:]<dev#>   the card & device number to play out on
    -f<frag_size>      requested fragment size
    -v                 verbose
    -s                 content is protected
    -e                 content would like to be played on a secure channel
    -r                 content can only be played on a secure channel
    -t                 print timing information of when data is sent in ns
    -w                 use separate threads to control and write audio data
    -c<args>[,args ..] voice matrix configuration
    -n<num_frags>      requested number of fragments
    -b<num_frags>      requested number of fragments while buffering
    -p<volume in %>    volume in percent
    -m<mixer name>     string name for mixer input
    -o<audio type>     name of the audio type registers with audioman
    -x                 use mmap interface
    -R<value>          SRC rate method
                       (1 = 7-pt kaiser windowed, 2 = 20-pt remez, 3 = linear interpolation)

Args:
    1=<hw_channel_bitmask> hardware channel bitmask for application voice 1
    2=<hw_channel_bitmask> hardware channel bitmask for application voice 2
    3=<hw_channel_bitmask> hardware channel bitmask for application voice 3
    4=<hw_channel_bitmask> hardware channel bitmask for application voice 4
    5=<hw_channel_bitmask> hardware channel bitmask for application voice 5
    6=<hw_channel_bitmask> hardware channel bitmask for application voice 6
    7=<hw_channel_bitmask> hardware channel bitmask for application voice 7
    8=<hw_channel_bitmask> hardware channel bitmask for application voice 8
#endif
/* *INDENT-ON* */
//*****************************************************************************

void sig_handler( int sig_no )
{
 running = false;
 return;
}

int
main (int argc, char **argv)
{
    int     card = -1;
    int     dev = 0;
    WriterData wd;
    WaveHdr wavHdr1;
    int     mSampleRate;
    int     mSampleChannels;
    int     mSampleBits;
    int     fragsize = -1;

    int     rtn;
    snd_pcm_channel_info_t pi;
    snd_pcm_channel_params_t pp;
    snd_pcm_channel_setup_t setup;
    int c;
    fd_set  rfds, wfds;
    uint32_t voice_mask[] = { 0, 0, 0, 0, 0, 0, 0, 0 };
    snd_pcm_voice_conversion_t voice_conversion;
```

```
    int     voice_override = 0;
    int     num_frags = -1;
    int     num_buffered_frags = 0;
    char   *sub_opts, *sub_opts_copy, *value;
    char   *dev_opts[] = {
#define CHN1 0
        "1",
#define CHN2 1
        "2",
#define CHN3 2
        "3",
#define CHN4 3
        "4",
#define CHN5 4
        "5",
#define CHN6 5
        "6",
#define CHN7 6
        "7",
#define CHN8 7
        "8",
        NULL
    };
    char    name[_POSIX_PATH_MAX] = { 0 };
    float   vol_percent = -1;
    float   volume;
    char    mixer_name[32];
    int     mix_name_enable = -1;
    int     protected_content = 0;
    int     enable_protection = 0;
    int     require_protection = 0;
    int     use_writer_thread = 0;
    int     uses_audioman_handle = 1;
    unsigned int audioman_handle;
    void   *retval;
    pthread_t writer_thread;
    pthread_t mixer_thread;
    pthread_t keypress_thread;
    char   *type = "multimedia";
    int    rate_method = 0;
    int    use_mmap = 0;

    while ((c = getopt (argc, argv, "a:ef:vc:n:b:p:m:qrstwo:xR:")) != EOF)
    {
        switch (c)
        {
        case 'a':
            if (strchr (optarg, ':'))
            {
                card = atoi (optarg);
                dev = atoi (strchr (optarg, ':') + 1);
            }
            else if (isalpha (optarg[0]) || optarg[0] == '/')
                strcpy (name, optarg);
            else
                dev = atoi (optarg);
            if (name[0] != '\0')
                printf ("Using device %s\n", name);
            else
                printf ("Using card %d device %d \n", card, dev);
            break;
        case 'f':
            fragsize = atoi (optarg);
            break;
        case 'v':
            verbose = 1;
            break;
        case 'c':
            sub_opts = sub_opts_copy = strdup (optarg);
            if (sub_opts == NULL) {
                printf("Cannot allocate sub_opts\n");
                exit(1);
            }
            while (*sub_opts != '\0')
            {
                int channel = getsubopt (&sub_opts, dev_opts, &value);
                if( channel >= 0 && channel < sizeof(voice_mask)/sizeof(voice_mask[0]) && value ) {
                    voice_mask[channel] = strtoul (value, NULL, 0);
                } else {
                    fprintf (stderr, "Invalid channel map specified\n");
```

```
                    exit(1);
                }
            }
            free(sub_opts_copy);
            voice_override = 1;
            break;
        case 'n':
            num_frags = atoi (optarg) - 1;
            break;
        case 'b':
            num_buffered_frags = atoi (optarg);
            break;
        case 'p':
            vol_percent = atof (optarg);
            break;
        case 'm':
            strncpy (mixer_name, optarg, 32);
            mix_name_enable = 1;
            break;
        case 's':
            protected_content = 1;
            break;
        case 'e':
            enable_protection = 1;
            break;
        case 'r':
            require_protection = 1;
            break;
        case 't':
            print_timing = 1;
            break;
        case 'w':
            use_writer_thread = 1;
            break;
        case 'o':
            type = optarg;
            break;
        case 'x':
            use_mmap = 1;
            break;
        case 'R':
            rate_method = atoi(optarg);
            if (rate_method < 0 || rate_method > 3)
            {
                rate_method = 0;
                printf("Invalid rate method, using method 0\n");
            }
            break;
        default:
            return 1;
        }
    }

    setvbuf (stdin, NULL, _IONBF, 0);

    if ( audio_manager_get_handle (audio_manager_get_type_from_name(type), 0, true, &audioman_handle) ) {
        uses_audioman_handle = 0;
    }
    if (name[0] != '\0')
    {
        snd_pcm_info_t info;

        if ((rtn = snd_pcm_open_name (&pcm_handle, name, SND_PCM_OPEN_PLAYBACK)) < 0)
        {
            return err ("open_name");
        }
        rtn = snd_pcm_info (pcm_handle, &info);
        card = info.card;
    }
    else
    {
        if (card == -1)
        {
            if ((rtn =
                    snd_pcm_open_preferred (&pcm_handle, &card, &dev, SND_PCM_OPEN_PLAYBACK)) < 0)
                return err ("device open");
        }
        else
        {
            if ((rtn = snd_pcm_open (&pcm_handle, card, dev, SND_PCM_OPEN_PLAYBACK)) < 0)
```



```
                return err ("device open");
        }
    }

    if( uses_audioman_handle ) {
        if ((rtn = snd_pcm_set_audioman_handle (pcm_handle, audioman_handle)) < 0)
            return err ("set audioman handle");
    }

    if (argc < 2)
        return err ("no file specified");

    if ((wd.file1 = fopen (argv[optind], "r")) == 0)
        return err ("file open #1");

    if (CheckHdr (wd.file1) == -1)
        return err ("CheckHdr #1");

    mSamples = FindTag (wd.file1, "fmt ");
    fread (&wavHdr1, sizeof (wavHdr1), 1, wd.file1);
    fseek (wd.file1, (mSamples - sizeof (WaveHdr)), SEEK_CUR);

    if( mBigEndian ) {
        mSampleRate = ENDIAN_BE32 (wavHdr1.SamplesPerSec);
        mSampleChannels = ENDIAN_BE16 (wavHdr1.Channels);
        mSampleBits = ENDIAN_BE16 (wavHdr1.BitsPerSample);
        wavHdr1.FormatTag = ENDIAN_BE16 (wavHdr1.FormatTag);
    } else {
        mSampleRate = ENDIAN_LE32 (wavHdr1.SamplesPerSec);
        mSampleChannels = ENDIAN_LE16 (wavHdr1.Channels);
        mSampleBits = ENDIAN_LE16 (wavHdr1.BitsPerSample);
        wavHdr1.FormatTag = ENDIAN_LE16 (wavHdr1.FormatTag);
    }

    printf ("SampleRate = %d, Channels = %d, SampleBits = %d\n", mSampleRate, mSampleChannels,
        mSampleBits);

    if (!use_mmap)
    {
        /* disabling mmap is not actually required in this example but it is included to
         * demonstrate how it is used when it is required.
         */
        if ((rtn = snd_pcm_plugin_set_disable (pcm_handle, PLUGIN_DISABLE_MMAP)) < 0)
        {
            fprintf (stderr, "snd_pcm_plugin_set_disable failed: %s\n", snd_strerror (rtn));
            return -1;
        }
    }

    if ((rtn = snd_pcm_plugin_set_enable (pcm_handle, PLUGIN_ROUTING)) < 0)
    {
        fprintf (stderr, "snd_pcm_plugin_set_enable failed for PLUGIN_ROUTING: %s\n", snd_strerror (rtn));
        return -1;
    }

    memset (&pi, 0, sizeof (pi));
    pi.channel = SND_PCM_CHANNEL_PLAYBACK;
    if ((rtn = snd_pcm_plugin_info (pcm_handle, &pi)) < 0)
    {
        fprintf (stderr, "snd_pcm_plugin_info failed: %s\n", snd_strerror (rtn));
        return -1;
    }

    memset (&pp, 0, sizeof (pp));

    pp.mode = SND_PCM_MODE_BLOCK
            | (protected_content ? SND_PCM_MODE_FLAG_PROTECTED_CONTENT : 0)
            | (enable_protection ? SND_PCM_MODE_FLAG_ENABLE_PROTECTION : 0)
            | (require_protection ? SND_PCM_MODE_FLAG_REQUIRE_PROTECTION : 0);

    pp.channel = SND_PCM_CHANNEL_PLAYBACK;
    pp.start_mode = SND_PCM_START_FULL;
    pp.stop_mode = SND_PCM_STOP_STOP;

    pp.buf.block.frag_size = pi.max_fragment_size;
    if (fragsize != -1)
    {
        pp.buf.block.frag_size = fragsize;
    }
    pp.buf.block.frags_max = num_frags;
```

```
pp.buf.block.frags_buffered_max = num_buffered_frags;
pp.buf.block.frags_min = 1;

pp.format.interleave = 1;
pp.format.rate = mSampleRate;
pp.format.voices = mSampleChannels;

if (wavHdr1.FormatTag == 6) {
    pp.format.format = SND_PCM_SFMT_A_LAW;
} else if (wavHdr1.FormatTag == 7) {
    pp.format.format = SND_PCM_SFMT_MU_LAW;
} else if (mSampleBits == 8) {
    pp.format.format = SND_PCM_SFMT_U8;
} else if (mSampleBits == 16) {
    if (mBigEndian) {
        pp.format.format = SND_PCM_SFMT_S16_BE;
    } else {
        pp.format.format = SND_PCM_SFMT_S16_LE;
    }
} else if (mSampleBits == 24) {
    if (mBigEndian) {
        pp.format.format = SND_PCM_SFMT_S24_BE;
    } else {
        pp.format.format = SND_PCM_SFMT_S24_LE;
    }
} else if (mSampleBits == 32) {
    if (mBigEndian) {
        pp.format.format = SND_PCM_SFMT_S32_BE;
    } else {
        pp.format.format = SND_PCM_SFMT_S32_LE;
    }
} else {
    fprintf(stderr, "Unsupported number of bits per sample %d", mSampleBits);
    return -1;
}

if (mix_name_enable == 1)
{
    strncpy (pp.sw_mixer_subchn_name, mixer_name, 32);
}
else
{
    strcpy (pp.sw_mixer_subchn_name, "Wave playback channel");
}

if ((rtn = snd_pcm_plugin_set_src_method(pcm_handle, rate_method)) != rate_method)
{
    fprintf(stderr, "Failed to apply rate_method %d, using %d\n", rate_method, rtn);
}

if ((rtn = snd_pcm_plugin_params (pcm_handle, &pp)) < 0)
{
    fprintf (stderr, "snd_pcm_plugin_params failed: %s, why_failed = %d\n", snd_strerror (rtn),
             pp.why_failed);
    return -1;
}

if ((rtn = snd_pcm_plugin_prepare (pcm_handle, SND_PCM_CHANNEL_PLAYBACK)) < 0)
    fprintf (stderr, "snd_pcm_plugin_prepare failed: %s\n", snd_strerror (rtn));

if (voice_override)
{
    snd_pcm_plugin_get_voice_conversion (pcm_handle, SND_PCM_CHANNEL_PLAYBACK,
        &voice_conversion);
    voice_conversion.matrix[0] = voice_mask[0];
    voice_conversion.matrix[1] = voice_mask[1];
    voice_conversion.matrix[2] = voice_mask[2];
    voice_conversion.matrix[3] = voice_mask[3];
    voice_conversion.matrix[4] = voice_mask[4];
    voice_conversion.matrix[5] = voice_mask[5];
    voice_conversion.matrix[6] = voice_mask[6];
    voice_conversion.matrix[7] = voice_mask[7];
    snd_pcm_plugin_set_voice_conversion (pcm_handle, SND_PCM_CHANNEL_PLAYBACK,
        &voice_conversion);
}

memset (&setup, 0, sizeof (setup));
memset (&group, 0, sizeof (group));
setup.channel = SND_PCM_CHANNEL_PLAYBACK;
setup.mixer_gid = &group.gid;
```

```
    if ((rtn = snd_pcm_plugin_setup (pcm_handle, &setup)) < 0)
    {
        fprintf (stderr, "snd_pcm_plugin_setup failed: %s\n", snd_strerror (rtn));
        return -1;
    }
    printf ("Format %s \n", snd_pcm_get_format_name (setup.format.format));
    printf ("Frag Size %d \n", setup.buf.block.frag_size);
    printf ("Total Frags %d \n", setup.buf.block.frags);
    printf ("Rate %d \n", setup.format.rate);
    printf ("Voices %d \n", setup.format.voices);
    bsize = setup.buf.block.frag_size;

    if (group.gid.name[0] == 0)
    {
        printf ("Mixer Pcm Group [%s] Not Set \n", group.gid.name);
    }
    else
    {
        printf ("Mixer Pcm Group [%s]\n", group.gid.name);
        if ((rtn = snd_mixer_open_pcm (&mixer_handle, pcm_handle)) < 0)
        {
            fprintf (stderr, "snd_mixer_open failed: %s\n", snd_strerror (rtn));
            return -1;
        }
    }
    if (tcgetpgrp (0) == getpid ())
        dev_raw (fileno (stdin));
    mSamples = FindTag (wd.file1, "data");

    if( print_timing ) {
        clock_gettime( CLOCK_REALTIME, &wd.start_time );
    }

    mSampleBfr1 = malloc (bsize);
    FD_ZERO (&rfds);
    FD_ZERO (&wfds);
    n = 1;

    if (mixer_handle)
    {
        if (vol_percent >=0)
        {
            if ((rtn = snd_mixer_group_read (mixer_handle, &group)) < 0)
                fprintf (stderr, "snd_mixer_group_read failed: %s\n", snd_strerror (rtn));

            volume = (float)(group.max - group.min) * ( vol_percent / 100);

            if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
                group.volume.names.front_left = (int)volume;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
                group.volume.names.rear_left = (int)volume;
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
                group.volume.names.front_center = (int)volume;
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
                group.volume.names.front_right = (int)volume;
            if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
                group.volume.names.rear_right = (int)volume;
            if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
                group.volume.names.woofer = (int)volume;

            if ((rtn = snd_mixer_group_write (mixer_handle, &group)) < 0)
                fprintf (stderr, "snd_mixer_group_write failed: %s\n", snd_strerror (rtn));

            vol_percent = -1;
        }
    }

    signal(SIGINT, sig_handler);
    signal(SIGTERM, sig_handler);

    if( use_writer_thread ) {
        pthread_create( &writer_thread, NULL, writer_thread_handler, &wd );
        pthread_create( &keypress_thread, NULL, generic_thread_handler, handle_keypress );
        if (mixer_handle)
            pthread_create( &mixer_thread, NULL, generic_thread_handler, handle_mixer );
        // First wait for feeder to complete. Any other thread will cause it to stop.
        // Then just kill the other threads
        pthread_join(writer_thread, &retval);
        pthread_cancel(keypress_thread);
        if (mixer_handle)
```

```
            pthread_cancel(mixer_thread);
    } else {
        while (running && N < mSamples && n > 0)
        {
            FD_ZERO(&rfds);
            FD_ZERO(&wfds);
            if (tcgetpgrp (0) == getpid ())
                FD_SET (STDIN_FILENO, &rfds);
            if (mixer_handle) {
                FD_SET (snd_mixer_file_descriptor (mixer_handle), &rfds);
            }
            FD_SET (snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_PLAYBACK), &wfds);

            rtn = max (snd_mixer_file_descriptor (mixer_handle),
                snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_PLAYBACK));

            if (select (rtn + 1, &rfds, &wfds, NULL, NULL) == -1)
            {
                err ("select");
                break; /* break loop to exit cleanly */
            }

            if (FD_ISSET (STDIN_FILENO, &rfds))
            {
                handle_keypress();
            }

            if (FD_ISSET (snd_mixer_file_descriptor (mixer_handle), &rfds))
            {
                handle_mixer();
            }

            if (FD_ISSET (snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_PLAYBACK), &wfds))
            {
                write_audio_data(&wd);
            }
        }
    }

    if (tcgetpgrp (0) == getpid ())
        dev_unraw (fileno (stdin));
    printf("Exiting...\n");
    if (running) {
        snd_pcm_plugin_flush (pcm_handle, SND_PCM_CHANNEL_PLAYBACK);
    }
    if (mixer_handle)
        snd_mixer_close (mixer_handle);
    snd_pcm_close (pcm_handle);

    if (uses_audioman_handle) {
        audio_manager_free_handle (audioman_handle);
    }
    fclose(wd.file1);

    return (0);
}
```

Appendix B
waverec.c example

This is a sample application that captures (i.e., records) audio data.

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */




#include <errno.h>
#include <fcntl.h>
#include <gulliver.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/stat.h>
#include <sys/termio.h>
#include <sys/types.h>
#include <unistd.h>
#include <limits.h>
#include <ctype.h>

#include <sys/asoundlib.h>

#include <audio/audio_manager_routing.h>

/* *INDENT-OFF* */
struct
{
    char        riff_id[4];
    uint32_t    wave_len;
    struct
    {
        char        fmt_id[8];
        uint32_t    fmt_len;
        struct
        {
            uint16_t    format_tag;
            uint16_t    voices;
            uint32_t    rate;
            uint32_t    char_per_sec;
            uint16_t    block_align;
            uint16_t    bits_per_sample;
        }
        fmt;
        struct
        {
            char        data_id[4];
            uint32_t    data_len;
        }
        data;
    }
    wave;
}
```

```
riff_hdr =
{
    {'R', 'I', 'F', 'F' },
    sizeof (riff_hdr.wave),
    {
        {'W', 'A', 'V', 'E', 'f', 'm', 't', ' '    },
        sizeof (riff_hdr.wave.fmt),
        {
            1, 0, 0, 0, 0, 0
        },
        {
            {'d', 'a', 't', 'a' },
            0,
        }
    }
};
/* *INDENT-ON* */


int
err (char *msg)
{
 perror (msg);
 return -1;
}


int
dev_raw (int fd)
{
 struct termios termios_p;

 if (tcgetattr (fd, &termios_p))
  return (-1);

 termios_p.c_cc[VMIN] = 1;
 termios_p.c_cc[VTIME] = 0;
 termios_p.c_lflag &= ~(ICANON | ECHO | ISIG);
 return (tcsetattr (fd, TCSANOW, &termios_p));
}

int
dev_unraw (int fd)
{
 struct termios termios_p;

 if (tcgetattr (fd, &termios_p))
  return (-1);

 termios_p.c_lflag |= (ICANON | ECHO | ISIG);
 return (tcsetattr (fd, TCSAFLUSH, &termios_p));
}


//*****************************************************************************
/* *INDENT-OFF* */
#ifdef __USAGE
%C[Options] *

Options:
    -8                 use 8 bit mode (16 bit default)
    -b <size>          Sample size (8, 16, 32)
    -a[card#:]<dev#>   the card & device number to record from
    -m                 record in mono (stereo default)
    -n <voices>        the number of voices to record (2 voices default, stereo)
    -r <rate>          record at rate (44100 default | 48000 44100 22050 11025)
    -t <sec>           seconds to record (5 seconds default)
    -f <frag_size>     requested fragment size
    -v                 verbosity
    -c <args>[,args ...] voice matrix configuration
    -o <audioman type> Specifies an audioman type for the capture stream.
    -x                 use mmap interface
    -i <0|1>           Interleave samples (default: 1)
    -R <value>         SRC rate method
                       (0 = linear interpolation, 1 = 7-pt kaiser windowed, 2 = 20-pt remez)
    -z<num_frags>      requested number of fragments

Note:
    If both 'm' and 'n' are specified in commandline, the one specified later will be used
```

```
Args:
    1=<hw_channel_bitmask> hardware channel bitmask for application voice 1
    2=<hw_channel_bitmask> hardware channel bitmask for application voice 2
    3=<hw_channel_bitmask> hardware channel bitmask for application voice 3
    4=<hw_channel_bitmask> hardware channel bitmask for application voice 4
#endif
/* *INDENT-ON* */
//*****************************************************************************

volatile int end = 0;

void sig_handler( int sig_no )
{
 end = 1;
 return;
}

const char *
why_failed ( int why_failed )
{
 switch (why_failed)
 {
  case SND_PCM_PARAMS_BAD_MODE:
   return ("Bad Mode Parameter");
  case SND_PCM_PARAMS_BAD_START:
   return ("Bad Start Parameter");
  case SND_PCM_PARAMS_BAD_STOP:
   return ("Bad Stop Parameter");
  case SND_PCM_PARAMS_BAD_FORMAT:
   return ("Bad Format Parameter");
  case SND_PCM_PARAMS_BAD_RATE:
   return ("Bad Rate Parameter");
  case SND_PCM_PARAMS_BAD_VOICES:
   return ("Bad Vocies Parameter");
  case SND_PCM_PARAMS_NO_CHANNEL:
   return ("No Channel Available");
  default:
   return ("Unknown Error");
 }

 return ("No Error");
}

int
main (int argc, char **argv)
{
 int     card = -1;
 int     dev = 0;
 int     ret;

 snd_pcm_t *pcm_handle;
 FILE   *file1;
 unsigned int mSamples;
 int     mSampleRate;
 int     mSampleChannels;
 int     mSampleBits;
 int     mSampleTime;
 char   *mSampleBfr1;
 int     fragsize = -1;
 int     num_frags = -1;
 int     verbose = 0;

 int     rtn;

 snd_pcm_channel_info_t pi;
 snd_mixer_t *mixer_handle = NULL;
 snd_mixer_group_t group;
 snd_pcm_channel_params_t pp;
 snd_pcm_channel_setup_t setup;
 int     bsize, N = 0, c;
 uint32_t voice_mask[] = { 0, 0, 0, 0 };
 snd_pcm_voice_conversion_t voice_conversion;
 int     voice_override = 0;
 char   *sub_opts, *value;
 char   *dev_opts[] = {
#define CHN1 0
  "1",
#define CHN2 1
  "2",
#define CHN3 2
```

```
  "3",
#define CHN4 3
  "4",
  NULL
 };
 char    name[_POSIX_PATH_MAX] = { 0 };
 char    *type = NULL;
 audio_manager_audio_type_t audioman_type;
 unsigned int audioman_handle;
 int     uses_audioman_handle = 0;
 int interleave = 1;
 fd_set  rfds;
 int use_mmap = 0;
 int rate_method = 0;

 mSampleRate = 44100;
 mSampleChannels = 2;
 mSampleBits = 16;
 mSampleTime = 5;
 while ((c = getopt (argc, argv, "8b:a:f:mn:r:t:vc:o:xi:R:z:")) != EOF)
 {
  switch (c)
  {
  case '8':
   mSampleBits = 8;
   break;
  case 'b':
   mSampleBits = atoi (optarg);
   if (mSampleBits != 8 && mSampleBits != 16 && mSampleBits != 24 && mSampleBits != 32)
   {
    printf("Invalid sample size, must be one of 8, 16, 24, 32\n");
    exit(1);
   }
   break;
  case 'a':
   if (strchr (optarg, ':'))
   {
    card = atoi (optarg);
    dev = atoi (strchr (optarg, ':') + 1);
   }
   else if (isalpha (optarg[0]) || optarg[0] == '/')
    strcpy (name, optarg);
   else
    dev = atoi (optarg);

   if (name[0] != '\0')
    printf ("Using device /dev/snd/%s\n", name);
   else
    printf ("Using card %d device %d \n", card, dev);
   break;
  case 'f':
   fragsize = atoi (optarg);
   break;
  case 'i':
   interleave = atoi(optarg);
   if (interleave <= 0)
    interleave = 0;
   else
    interleave = 1;
   break;
  case 'm':
   mSampleChannels = 1;
   break;
  case 'n':
   mSampleChannels = atoi (optarg);
   break;
  case 'r':
   mSampleRate = atoi (optarg);
   break;
  case 't':
   mSampleTime = atoi (optarg);
   break;
  case 'v':
   verbose = 1;
   break;
  case 'c':
   sub_opts = strdup (optarg);
   while (*sub_opts != '\0')
   {
    switch (getsubopt (&sub_opts, dev_opts, &value))
```

```
    {
    case CHN1:
     voice_mask[0] = strtoul (value, NULL, 0);
     break;
    case CHN2:
     voice_mask[1] = strtoul (value, NULL, 0);
     break;
    case CHN3:
     voice_mask[2] = strtoul (value, NULL, 0);
     break;
    case CHN4:
     voice_mask[3] = strtoul (value, NULL, 0);
     break;
    default:
     break;
    }
   }
   voice_override = 1;
   break;
  case 'o':
   type = optarg;
   uses_audioman_handle = 1;
   break;
  case 'x':
   use_mmap = 1;
   break;
  case 'R':
   rate_method = atoi(optarg);
   if (rate_method < 0 || rate_method > 2)
   {
    rate_method = 0;
    printf("Invalid rate method, using method 0\n");
   }
   break;
  case 'z':
   num_frags = atoi (optarg) - 1;
   break;
  default:
   return 1;
  }
 }

 // Setup audioman handle if it is available
 if (uses_audioman_handle) {
  audioman_type = audio_manager_get_type_from_name( type );
  if ( audio_manager_get_handle( audioman_type,
                                 getpid(),
                                 false,
                                 &audioman_handle ) >= 0) {
   uses_audioman_handle = 1;
  } else {
   // Return an error because the user wanted to use audioman.
   return err("Audio Manager is not available");
  }
 }

 if (name[0] != '\0')
 {
  snd_pcm_info_t info;

  if ((rtn = snd_pcm_open_name (&pcm_handle, name, SND_PCM_OPEN_CAPTURE)) < 0)
  {
   return err ("open_name");
  }
  rtn = snd_pcm_info (pcm_handle, &info);
  card = info.card;
 }
 else
 {
  if (card == -1)
  {
   if ((rtn = snd_pcm_open_preferred (&pcm_handle, &card, &dev, SND_PCM_OPEN_CAPTURE)) < 0)
    return err ("device open");
  }
  else
  {
   if ((rtn = snd_pcm_open (&pcm_handle, card, dev, SND_PCM_OPEN_CAPTURE)) < 0)
    return err ("device open");
  }
 }
```

```
if (optind >= argc)
 return err ("no file specified");

if ((file1 = fopen (argv[optind], "w")) == 0)
 return err ("file open #1");

if( mSampleTime == 0 ) {
 mSamples = 0xFFFFFFFF - sizeof(riff_hdr) + 8;
} else {
 mSamples = mSampleRate * mSampleChannels * mSampleBits / 8 * mSampleTime;
}

if (uses_audioman_handle) {
 snd_pcm_set_audioman_handle( pcm_handle, audioman_handle );
 if(type) {
  printf("Audio Manager Type: %s\n", type);
 }
}

riff_hdr.wave.fmt.voices = ENDIAN_LE16 (mSampleChannels);
riff_hdr.wave.fmt.rate = ENDIAN_LE32 (mSampleRate);
riff_hdr.wave.fmt.char_per_sec =
 ENDIAN_LE32 (mSampleRate * mSampleChannels * mSampleBits / 8);
riff_hdr.wave.fmt.block_align = ENDIAN_LE16 (mSampleChannels * mSampleBits / 8);
riff_hdr.wave.fmt.bits_per_sample = ENDIAN_LE16 (mSampleBits);
riff_hdr.wave.data.data_len = ENDIAN_LE32 (mSamples);
riff_hdr.wave_len = ENDIAN_LE32 (mSamples + sizeof (riff_hdr) - 8);
fwrite (&riff_hdr, 1, sizeof (riff_hdr), file1);

printf ("SampleRate = %d, Channels = %d, SampleBits = %d\n", mSampleRate, mSampleChannels,
 mSampleBits);

if (!use_mmap)
{
 /* disabling mmap is not actually required in this example but it is included to
  * demonstrate how it is used when it is required.
  */
 if ((rtn = snd_pcm_plugin_set_disable (pcm_handle, PLUGIN_DISABLE_MMAP)) < 0)
 {
  fprintf (stderr, "snd_pcm_plugin_set_disable failed: %s\n", snd_strerror (rtn));
  return -1;
 }
}

memset (&pi, 0, sizeof (pi));
pi.channel = SND_PCM_CHANNEL_CAPTURE;
if ((rtn = snd_pcm_plugin_info (pcm_handle, &pi)) < 0)
{
 fprintf (stderr, "snd_pcm_plugin_info failed: %s\n", snd_strerror (rtn));
 return -1;
}

memset (&pp, 0, sizeof (pp));

pp.mode = SND_PCM_MODE_BLOCK;
pp.channel = SND_PCM_CHANNEL_CAPTURE;
pp.start_mode = SND_PCM_START_DATA;
pp.stop_mode = SND_PCM_STOP_STOP;
pp.time = 1;

pp.buf.block.frag_size = pi.max_fragment_size;
if (fragsize != -1)
 pp.buf.block.frag_size = fragsize;
pp.buf.block.frags_max = num_frags;
pp.buf.block.frags_min = 1;

pp.format.interleave = interleave;
pp.format.rate = mSampleRate;
pp.format.voices = mSampleChannels;

switch (mSampleBits)
{
 case 8:
  pp.format.format = SND_PCM_SFMT_U8;
  break;
 case 16:
 default:
  pp.format.format = SND_PCM_SFMT_S16_LE;
  break;
```

```
 case 24:
  pp.format.format = SND_PCM_SFMT_S24_LE;
  break;
 case 32:
  pp.format.format = SND_PCM_SFMT_S32_LE;
  break;
}

if ((rtn = snd_pcm_plugin_set_src_method(pcm_handle, rate_method)) != rate_method)
{
 fprintf(stderr, "Failed to apply rate_method %d, using %d\n", rate_method, rtn);
}

if ((rtn = snd_pcm_plugin_params (pcm_handle, &pp)) < 0)
{
 fprintf (stderr, "snd_pcm_plugin_params failed: %s - %s\n", snd_strerror (rtn), why_failed(pp.why_failed));
 return -1;
}

if ((rtn = snd_pcm_plugin_prepare (pcm_handle, SND_PCM_CHANNEL_CAPTURE)) < 0)
 fprintf (stderr, "snd_pcm_plugin_prepare failed: %s\n", snd_strerror (rtn));


if (voice_override)
{
 snd_pcm_plugin_get_voice_conversion (pcm_handle, SND_PCM_CHANNEL_CAPTURE,
  &voice_conversion);
 voice_conversion.matrix[0] = voice_mask[0];
 voice_conversion.matrix[1] = voice_mask[1];
 voice_conversion.matrix[2] = voice_mask[2];
 voice_conversion.matrix[3] = voice_mask[3];
 snd_pcm_plugin_set_voice_conversion (pcm_handle, SND_PCM_CHANNEL_CAPTURE,
  &voice_conversion);
}

memset (&setup, 0, sizeof (setup));
memset (&group, 0, sizeof (group));
setup.channel = SND_PCM_CHANNEL_CAPTURE;
setup.mixer_gid = &group.gid;
if ((rtn = snd_pcm_plugin_setup (pcm_handle, &setup)) < 0)
{
 fprintf (stderr, "snd_pcm_plugin_setup failed: %s\n", snd_strerror (rtn));
 return -1;
}
printf ("Format %s \n", snd_pcm_get_format_name (setup.format.format));
printf ("Frag Size %d \n", setup.buf.block.frag_size);
printf ("Total Frags %d \n", setup.buf.block.frags);
printf ("Rate %d \n", setup.format.rate);
bsize = setup.buf.block.frag_size;

if (group.gid.name[0] == 0)
{
 printf ("Mixer Pcm Group [%s] Not Set \n", group.gid.name);
 printf ("***>>>> Input Gain Controls Disabled <<<<*** \n");
}
else
{
 printf ("Mixer Pcm Group [%s]\n", group.gid.name);
 if ((rtn = snd_mixer_open (&mixer_handle, card, setup.mixer_device)) < 0)
 {
  fprintf (stderr, "snd_mixer_open failed: %s\n", snd_strerror (rtn));
  return -1;
 }
}

if (tcgetpgrp (0) == getpid ())
 dev_raw (fileno (stdin));

mSampleBfr1 = malloc (bsize);
FD_ZERO (&rfds);
signal(SIGINT, sig_handler);
signal(SIGTERM, sig_handler);
while (!end && N < mSamples)
{
 if (mixer_handle)
 {
  /* If we are the foreground process group associated with STDIN then include
   * STDIN in the fdset to handle user volume adjustments.
   */
  if (tcgetpgrp (0) == getpid ())
```

```
            FD_SET (STDIN_FILENO, &rfds);
           /* Include the mixer_handle descriptor in the fdset to handle
            * mixer events.
            */
           FD_SET (snd_mixer_file_descriptor (mixer_handle), &rfds);
          }
          FD_SET (snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_CAPTURE), &rfds);

          rtn = max (snd_mixer_file_descriptor (mixer_handle),
           snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_CAPTURE));

          if (select (rtn + 1, &rfds, NULL, NULL, NULL) == -1)
          {
           err ("select");
           break; /* break loop to exit cleanly */
          }

          if (FD_ISSET (STDIN_FILENO, &rfds))
          {
           c = getc (stdin);
           if (c != EOF)
           {
            if (group.gid.name[0] != 0)
            {
             if ((rtn = snd_mixer_group_read (mixer_handle, &group)) < 0)
              fprintf (stderr, "snd_mixer_group_read failed: %s\n", snd_strerror (rtn));
             switch (c)
             {
             case 'q':
              group.volume.names.front_left += 1;
              break;
             case 'a':
              group.volume.names.front_left -= 1;
              break;
             case 'w':
              group.volume.names.front_left += 1;
              group.volume.names.front_right += 1;
              break;
             case 's':
              group.volume.names.front_left -= 1;
              group.volume.names.front_right -= 1;
              break;
             case 'e':
              group.volume.names.front_right += 1;
              break;
             case 'd':
              group.volume.names.front_right -= 1;
              break;
             case 'o':
              sleep(5);
              break;
             case 'f':
              if( (ret = snd_pcm_plugin_flush( pcm_handle, SND_PCM_CHANNEL_CAPTURE )) == 0 ) {
               printf("Flushing\n");
              } else {
               fprintf(stderr, "Flush failed: %d\n", ret);
              }
              break;
             case 'g':
              if( (ret = snd_pcm_plugin_prepare( pcm_handle, SND_PCM_CHANNEL_CAPTURE )) == 0 ) {
               printf("Preparing\n");
              } else {
               fprintf(stderr, "Preparing failed: %d\n", ret);
              }
              break;
             case 'p':
              if( (ret = snd_pcm_capture_pause( pcm_handle )) == 0 ) {
               printf("Pausing\n");
              } else {
               fprintf(stderr, "Pause failed: %d\n", ret);
              }
              break;
             case 'r':
              if( (ret = snd_pcm_capture_resume( pcm_handle )) == 0 ) {
               printf("Resuming\n");
              } else {
               fprintf(stderr, "Resume failed: %d\n", ret);
              }
              break;
             case 3: //Ctrl-C
```

```
        case 27: // Escape
         end = 1;
         break;
        case 'z':
         printf("delaying 500ms\n");
         delay(500);
         break;
        }
        if (group.volume.names.front_left > group.max)
         group.volume.names.front_left = group.max;
        if (group.volume.names.front_left < group.min)
         group.volume.names.front_left = group.min;
        if (group.volume.names.front_right > group.max)
         group.volume.names.front_right = group.max;
        if (group.volume.names.front_right < group.min)
         group.volume.names.front_right = group.min;
        if ((rtn = snd_mixer_group_write (mixer_handle, &group)) < 0)
         fprintf (stderr, "snd_mixer_group_write failed: %s\n", snd_strerror (rtn));

        if (group.max==group.min)
         printf ("Volume Now at %d:%d\n", group.max, group.max);
        else
         printf ("Volume Now at %d:%d \n",
         100 * (group.volume.names.front_left - group.min) / (group.max - group.min),
         100 * (group.volume.names.front_right - group.min) / (group.max -
          group.min));
       }
      }
      else {
       if (tcgetpgrp (0) == getpid ())
        dev_unraw (fileno (stdin));
       exit (0);
      }
     }

     if (FD_ISSET (snd_mixer_file_descriptor (mixer_handle), &rfds))
     {
      snd_mixer_callbacks_t callbacks = {
       0, 0, 0, 0
      };

      snd_mixer_read (mixer_handle, &callbacks);
     }

     if (FD_ISSET (snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_CAPTURE), &rfds))
     {
      snd_pcm_channel_status_t status;
      int     read = 0;

      read = snd_pcm_plugin_read (pcm_handle, mSampleBfr1, bsize);
      if (verbose)
       printf ("bytes read = %d, bsize = %d \n", read, bsize);
      if (read < bsize)
      {
       memset (&status, 0, sizeof (status));
       status.channel = SND_PCM_CHANNEL_CAPTURE;
       if (snd_pcm_plugin_status (pcm_handle, &status) < 0)
       {
        fprintf (stderr, "Capture channel status error\n");
        exit (1);
       }

       if (status.status == SND_PCM_STATUS_READY ||
        status.status == SND_PCM_STATUS_OVERRUN)
       {
        if (status.status == SND_PCM_STATUS_OVERRUN)
         fprintf(stderr, "overrun: capture channel\n");

        if (snd_pcm_plugin_prepare (pcm_handle, SND_PCM_CHANNEL_CAPTURE) < 0)
        {
         fprintf (stderr, "Capture channel prepare error\n");
         exit (1);
        }
       }
       else if (status.status == SND_PCM_STATUS_ERROR)
       {
        fprintf(stderr, "error: capture channel failure\n");
        exit(1);
       }
       else if (status.status == SND_PCM_STATUS_CHANGE)
```

```
        {
         fprintf(stderr, "change: capture channel capability change\n");
         exit(1);
        }
        else if (status.status == SND_PCM_STATUS_PREEMPTED)
        {
         fprintf(stderr, "error: capture channel preempted\n");
         exit(1);
        }
       } else {
        fwrite (mSampleBfr1, 1, read, file1);
        N += read;
       }
      }
     }

     if (tcgetpgrp (0) == getpid ())
      dev_unraw (fileno (stdin));

     printf("Exiting...\n");
     snd_pcm_plugin_flush (pcm_handle, SND_PCM_CHANNEL_CAPTURE);

     rtn = snd_mixer_close (mixer_handle);
     rtn = snd_pcm_close (pcm_handle);
     fclose (file1);

     if (uses_audioman_handle) {
      audio_manager_free_handle( audioman_handle );
     }
     return (0);
    }
```

Appendix C

mix_ctl.c example

This is a sample application that captures the groups and switches in the mixer.

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */




#include <errno.h>
#include <fcntl.h>
#include <fnmatch.h>
#include <gulliver.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/stat.h>
#include <sys/termio.h>
#include <sys/types.h>
#include <unistd.h>

#include <sys/asoundlib.h>


//*****************************************************************************
/* *INDENT-OFF* */
#ifdef __USAGE
%C  [Options] Cmds

Options:
 -a[card#:]<dev#>  the card & mixer device number to access

Cmds:

 groups  [-d] [-c] [-p] [pattern]
  -d  will print the group details
  -c  will show only groups effecting capture
  -p  will show only groups effecting playback

 group  name  [mute[Y]=off|on]  [capture[Y]=off|on]  [volume[Y]=x|x%]  ...
  - name is the group name quoted if it contains white space
  - the Y is a option the restricts the change to only one voice (if possible)

 switches  [pattern]

 switch  name  [value]
  - name is the switch name quoted if it contains white space

#endif
/* *INDENT-ON* */
//*****************************************************************************


void
```

```
display_group (snd_mixer_t * mixer_handle, snd_mixer_gid_t * gid, snd_mixer_group_t * group)
{
 int     j;

 printf ("\"%s\",%d - %s \n", gid->name, gid->index,
  group->caps & SND_MIXER_GRPCAP_PLAY_GRP ? "Playback Group" : "Capture Group");

 printf ("\tCapabilities - ");
 if (group->caps & SND_MIXER_GRPCAP_VOLUME)
  printf (" Volume");
 if (group->caps & SND_MIXER_GRPCAP_JOINTLY_MUTE)
  printf (" Jointly-Mute");
 else if (group->caps & SND_MIXER_GRPCAP_MUTE)
  printf (" Mute");
 if (group->caps & SND_MIXER_GRPCAP_JOINTLY_CAPTURE)
  printf (" Jointly-Capture");
 if (group->caps & SND_MIXER_GRPCAP_EXCL_CAPTURE)
  printf (" Exclusive-Capture");
 else if (group->caps & SND_MIXER_GRPCAP_CAPTURE)
  printf (" Capture");
 printf ("\n");

 printf ("\tChannels - ");
 for (j = 0; j <= SND_MIXER_CHN_LAST; j++)
 {
  if (!(group->channels & (1 << j)))
   continue;
  printf ("%s ", snd_mixer_channel_name (j));
 }
 printf ("\n");

 printf ("\tVolume Range - minimum=%i, maximum=%i\n", group->min, group->max);

 for (j = 0; j <= SND_MIXER_CHN_LAST; j++)
 {
  if (!(group->channels & (1 << j)))
   continue;
  printf ("\tChannel %d %-12.12s - %3d (%3d%%) %s %s\n", j,
   snd_mixer_channel_name (j), group->volume.values[j],
   (group->max - group->min) <= 0 ? 0 : 100 * (group->volume.values[j] - group->min)
   / (group->max - group->min),
   group->mute & (1 << j) ? "Muted" : "", group->capture & (1 << j) ? "Capture" : "");
 }
}


void
display_groups (snd_mixer_t * mixer_handle, int argc, char *argv[])
{
 char    details = 0;
 char    playback_only = 0, capture_only = 0;
 char   *pattern;
 snd_mixer_groups_t groups;
 int     i;
 int     rtn;
 snd_mixer_group_t group;

 optind = 1;
 while ((i = getopt (argc, argv, "cdp")) != EOF)
 {
  switch (i)
  {
  case 'c':
   capture_only = 1;
   playback_only = 0;
   break;
  case 'd':
   details = 1;
   break;
  case 'p':
   capture_only = 0;
   playback_only = 1;
   break;
  }
 }
 pattern = (optind >= argc) ? "*" : argv[optind];

 while (1)
 {
  memset (&groups, 0, sizeof (groups));
```

```
    if (snd_mixer_groups (mixer_handle, &groups) < 0)
    {
     fprintf (stderr, "snd_mixer_groups API call - %s", strerror (errno));
    }
    else if (groups.groups == 0)
    {
     fprintf (stderr, "--> No mixer groups to list <-- \n");
     break;
    }

    if (groups.groups_over > 0)
    {
     groups.groups_size = groups.groups_over;
     groups.pgroups =
      (snd_mixer_gid_t *) malloc (sizeof (snd_mixer_gid_t) * groups.groups_size);
     if (groups.pgroups == NULL) {
      fprintf (stderr, "Unable to malloc group array - %s", strerror (errno));
      break;
     }
     groups.groups_over = 0;
     groups.groups = 0;
     if (snd_mixer_groups (mixer_handle, &groups) < 0)
      fprintf (stderr, "No Mixer Groups ");
     if (groups.groups_over > 0)
     {
      free (groups.pgroups);
      continue;
     }
     else
     {
      snd_mixer_sort_gid_table (groups.pgroups, groups.groups_size,
       snd_mixer_default_weights);
      break;
     }
    }
   }

   for (i = 0; i < groups.groups; i++)
   {
    if (fnmatch (pattern, groups.pgroups[i].name, 0) == 0)
    {
     memset (&group, 0, sizeof (group));
     memcpy (&group.gid, &groups.pgroups[i], sizeof (snd_mixer_gid_t));
     if ((rtn = snd_mixer_group_read (mixer_handle, &group)) < 0)
      fprintf (stderr, "snd_mixer_group_read failed: %s\n", snd_strerror (rtn));

     if (playback_only && group.caps & SND_MIXER_GRPCAP_CAP_GRP)
      continue;
     if (capture_only && group.caps & SND_MIXER_GRPCAP_PLAY_GRP)
      continue;

     if (details)
      display_group (mixer_handle, &groups.pgroups[i], &group);
     else
     {
      printf ("\"%s\",%d%*c - %s \n",
       groups.pgroups[i].name, groups.pgroups[i].index,
       2 + sizeof (groups.pgroups[i].name) - strlen (groups.pgroups[i].name), ' ',
       group.caps & SND_MIXER_GRPCAP_PLAY_GRP ? "Playback Group" : "Capture Group");
     }
    }
   }
  }


  int
  find_group_best_match (snd_mixer_t * mixer_handle, snd_mixer_gid_t * gid, snd_mixer_group_t * group)
  {
   snd_mixer_groups_t groups;
   int     i;

   while (1)
   {
    memset (&groups, 0, sizeof (groups));
    if (snd_mixer_groups (mixer_handle, &groups) < 0)
    {
     fprintf (stderr, "snd_mixer_groups API call - %s", strerror (errno));
    }
    if (groups.groups_over > 0)
    {
```

```
      groups.groups_size = groups.groups_over;
      groups.pgroups =
       (snd_mixer_gid_t *) malloc (sizeof (snd_mixer_gid_t) * groups.groups_size);
      if (groups.pgroups == NULL)
       fprintf (stderr, "Unable to malloc group array - %s", strerror (errno));
      groups.groups_over = 0;
      groups.groups = 0;
      if (snd_mixer_groups (mixer_handle, &groups) < 0)
       fprintf (stderr, "No Mixer Groups ");
      if (groups.groups_over > 0)
      {
       free (groups.pgroups);
       continue;
      }
      else
       break;
     }
    }

    for (i = 0; i < groups.groups; i++)
    {
     if (stricmp (gid->name, groups.pgroups[i].name) == 0 &&
      gid->index == groups.pgroups[i].index)
     {
      memset (group, 0, sizeof (group));
      memcpy (gid, &groups.pgroups[i], sizeof (snd_mixer_gid_t));
      memcpy (&group->gid, &groups.pgroups[i], sizeof (snd_mixer_gid_t));
      if ((snd_mixer_group_read (mixer_handle, group)) < 0)
       return ENOENT;
      else
       return EOK;
     }
    }

    return ENOENT;
   }


   int
   group_option_value (char *option)
   {
    char   *ptr;
    int     value;

    if ((ptr = strrchr (option, '=')) != NULL)
    {
     if (*(ptr + 1) == 0)
      value = -2;
     else if (stricmp (ptr + 1, "off") == 0)
      value = 0;
     else if (stricmp (ptr + 1, "on") == 0)
      value = 1;
     else
      value = atoi (ptr + 1);
    }
    else
     value = -1;
    return (value);
   }

   void
   modify_group (snd_mixer_t * mixer_handle, int argc, char *argv[])
   {
    int     optind = 1;
    snd_mixer_gid_t gid;
    char   *ptr;
    int     rtn;
    snd_mixer_group_t group;
    uint32_t channel = 0, j;
    int32_t value;
    char    modified = 0;

    if (optind >= argc)
    {
     fprintf (stderr, "No Group secified \n");
     return;
    }

    memset (&gid, 0, sizeof (gid));
    ptr = strtok (argv[optind++], ",");
```



```
if (ptr != NULL) {
 strncpy (gid.name, ptr, sizeof (gid.name));
 ptr = strtok (NULL, " ");
   }
if (ptr != NULL)
 gid.index = atoi (ptr);

memset (&group, 0, sizeof (group));
memcpy (&group.gid, &gid, sizeof (snd_mixer_gid_t));
if ((rtn = snd_mixer_group_read (mixer_handle, &group)) < 0)
{
 if (rtn == -ENXIO)
  rtn = find_group_best_match (mixer_handle, &gid, &group);

 if (rtn != EOK)
 {
  fprintf (stderr, "snd_mixer_group_read failed: %s\n", snd_strerror (rtn));
  return;
 }
}

/* if we have a value option set the group, write and reread it (to get true driver state) */
/* some things like capture (MUX) can't be turned off but can only be set on another group */
while (optind < argc)
{
 modified = 1;
 if ((value = group_option_value (argv[optind])) < 0)
  printf ("\n\t>>>> Unrecognized option [%s] <<<<\n\n", argv[optind]);
 else if (strnicmp (argv[optind], "mute", 4) == 0)
 {
  if (argv[optind][4] == '=')
   channel = LONG_MAX;
  else
   channel = atoi (&argv[optind][4]);
  if (channel == LONG_MAX)
   group.mute = value ? LONG_MAX : 0;
  else
  {
   group.mute = value ? group.mute | (1 << channel) : group.mute & ~(1 << channel);
  }
 }
 else if (strnicmp (argv[optind], "capture", 7) == 0)
 {
  if (argv[optind][7] == '=')
   channel = LONG_MAX;
  else
   channel = atoi (&argv[optind][7]);
  if (channel == LONG_MAX)
   group.capture = value ? LONG_MAX : 0;
  else
   group.capture =
    value ? group.capture | (1 << channel) : group.capture & ~(1 << channel);
 }
 else if (strnicmp (argv[optind], "volume", 6) == 0)
 {
  if (argv[optind][6] == '=')
   channel = LONG_MAX;
  else
   channel = atoi (&argv[optind][6]);
  if (argv[optind][strlen (argv[optind]) - 1] == '%' && (group.max - group.min) >= 0)
   value = (value * (group.max - group.min)) / 100 + group.min;
  if (value > group.max)
   value = group.max;
  if (value < group.min)
   value = group.min;
  for (j = 0; j <= SND_MIXER_CHN_LAST; j++)
  {
   if (!(group.channels & (1 << j)))
    continue;
   if (channel == LONG_MAX || channel == j)
    group.volume.values[j] = value;
  }
 }
 else if (strnicmp (argv[optind], "delay", 5) == 0)
 {
  if (argv[optind][5] == '=')
   group.change_duration = value;
  else
   group.change_duration = 50000;
 }
```

```
   else
    printf ("\n\t>>>> Unrecognized option [%s] <<<<\n\n", argv[optind]);

   if (channel != LONG_MAX && !(group.channels & (1 << channel)))
    printf ("\n\t>>>> Channel specified [%d] Not in group <<<<\n\n", channel);
   optind++;
  }

  if (modified)
   if ((rtn = snd_mixer_group_write (mixer_handle, &group)) < 0)
    fprintf (stderr, "snd_mixer_group_write failed: %s\n", snd_strerror (rtn));
  if ((rtn = snd_mixer_group_read (mixer_handle, &group)) < 0)
   fprintf (stderr, "snd_mixer_group_read failed: %s\n", snd_strerror (rtn));

  /* display the current group state */
  display_group (mixer_handle, &gid, &group);
 }


 void
 display_switch (snd_switch_t * sw, char table_formated)
 {
  printf ("\"%s\"%*c ", sw->name,
   table_formated ? sizeof (sw->name) - strlen (sw->name) : 1, ' ');
  switch (sw->type)
  {
  case SND_SW_TYPE_BOOLEAN:
   printf ("%s  %s \n", "BOOLEAN", sw->value.enable ? "on" : "off");
   break;
  case SND_SW_TYPE_BYTE:
   printf ("%s  %d \n", "BYTE   ", sw->value.byte.data);
   break;
  case SND_SW_TYPE_WORD:
   printf ("%s  %d \n", "WORD   ", sw->value.word.data);
   break;
  case SND_SW_TYPE_DWORD:
   printf ("%s  %d \n", "DWORD  ", sw->value.dword.data);
   break;
  case SND_SW_TYPE_LIST:
   if (sw->subtype == SND_SW_SUBTYPE_HEXA)
    printf ("%s  0x%x \n", "LIST   ", sw->value.list.data);
   else
    printf ("%s  %d \n", "LIST   ", sw->value.list.data);
   break;
  case SND_SW_TYPE_STRING_11:
   printf ("%s  \"%s\" \n", "STRING ",
    sw->value.string_11.strings[sw->value.string_11.selection]);
   break;
  default:
   printf ("%s  %d \n", "?      ", 0);
  }
 }


 void
 display_switches (snd_ctl_t * ctl_handle, int mixer_dev, int argc, char *argv[])
 {
  int     i;
  char   *pattern;
  snd_switch_list_t list;
  snd_switch_t sw;
  int     rtn;

  optind = 1;
  while ((i = getopt (argc, argv, "d")) != EOF)
  {
   switch (i)
   {
   }
  }
  pattern = (optind >= argc) ? "*" : argv[optind];
  while (1)
  {
   memset (&list, 0, sizeof (list));
   if (snd_ctl_mixer_switch_list (ctl_handle, mixer_dev, &list) < 0)
   {
    fprintf (stderr, "snd_ctl_mixer_switch_list API call - %s", strerror (errno));
   }
   else if (list.switches == 0)
   {
```

```
      fprintf (stderr, "--> No mixer switches to list <-- \n");
      break;
     }

     if (list.switches_over > 0)
     {
      list.switches_size = list.switches_over;
      list.pswitches = malloc (sizeof (snd_switch_list_item_t) * list.switches_size);
      if (list.pswitches == NULL)
       fprintf (stderr, "Unable to malloc switch array - %s", strerror (errno));
      list.switches_over = 0;
      list.switches = 0;
      if (snd_ctl_mixer_switch_list (ctl_handle, mixer_dev, &list) < 0)
       fprintf (stderr, "No Switches ");
      if (list.switches_over > 0)
      {
       free (list.pswitches);
       continue;
      }
      else
       break;
     }
    }

    for (i = 0; i < list.switches_size; i++)
    {
     memset (&sw, 0, sizeof (sw));
     strncpy (sw.name, (&list.pswitches[i])->name, sizeof (sw.name));
     if ((rtn = snd_ctl_mixer_switch_read (ctl_handle, mixer_dev, &sw)) < 0)
      fprintf (stderr, "snd_ctl_mixer_switch_read failed: %s\n", snd_strerror (rtn));
     display_switch (&sw, 1);
    }
   }


   void
   modify_switch (snd_ctl_t * ctl_handle, int mixer_dev, int argc, char *argv[])
   {
    int     optind = 1;
    snd_switch_t sw;
    int     rtn;
    int     value = 0;
    char   *string = NULL;

    if (optind >= argc)
    {
     fprintf (stderr, "No Switch secified \n");
     return;
    }

    memset (&sw, 0, sizeof (sw));
    strncpy (sw.name, argv[optind++], sizeof (sw.name));
    if ((rtn = snd_ctl_mixer_switch_read (ctl_handle, mixer_dev, &sw)) < 0)
    {
     fprintf (stderr, "snd_ctl_mixer_switch_read failed: %s\n", snd_strerror (rtn));
     return;
    }

    /* if we have a value option set the sw, write and reread it (to get true driver state) */
    if (optind < argc)
    {
     if (stricmp (argv[optind], "off") == 0)
      value = 0;
     else if (stricmp (argv[optind], "on") == 0)
      value = 1;
     else if (strnicmp (argv[optind], "0x", 2) == 0)
      value = strtol (argv[optind], NULL, 16);
     else
     {
      value = atoi (argv[optind]);
      string = argv[optind];
     }
     optind++;
     if (sw.type == SND_SW_TYPE_BOOLEAN)
      sw.value.enable = value;
     else if (sw.type == SND_SW_TYPE_BYTE)
      sw.value.byte.data = value;
     else if (sw.type == SND_SW_TYPE_WORD)
      sw.value.word.data = value;
     else if (sw.type == SND_SW_TYPE_DWORD)
```

```
        sw.value.dword.data = value;
       else if (sw.type == SND_SW_TYPE_LIST)
        sw.value.list.data = value;
       else if (sw.type == SND_SW_TYPE_STRING_11)
       {
        for (rtn = 0; rtn < sw.value.string_11.strings_cnt; rtn++)
        {
         if (stricmp (string, sw.value.string_11.strings[rtn]) == 0)
         {
          sw.value.string_11.selection = rtn;
          break;
         }
        }
        if (rtn == sw.value.string_11.strings_cnt)
        {
         fprintf (stderr, "ERROR string \"%s\" NOT IN LIST \n", string);
         snd_ctl_mixer_switch_read (ctl_handle, mixer_dev, &sw);
        }
       }
       if ((rtn = snd_ctl_mixer_switch_write (ctl_handle, mixer_dev, &sw)) < 0)
        fprintf (stderr, "snd_ctl_mixer_switch_write failed: %s\n", snd_strerror (rtn));
       if ((rtn = snd_ctl_mixer_switch_read (ctl_handle, mixer_dev, &sw)) < 0)
        fprintf (stderr, "snd_ctl_mixer_switch_read failed: %s\n", snd_strerror (rtn));
      }

      /* display the current switch state */
      display_switch (&sw, 0);
     }


     int
     main (int argc, char *argv[])
     {
      int     c;
      int     card = 0;
      int     dev = 0;
      int     rtn;
      snd_ctl_t *ctl_handle;
      snd_mixer_t *mixer_handle;

      optind = 1;
      while ((c = getopt (argc, argv, "a:")) != EOF)
      {
       switch (c)
       {
       case 'a':
        if (strchr (optarg, ':'))
        {
         card = atoi (optarg);
         dev = atoi (strchr (optarg, ':') + 1);
        }
        else
         dev = atoi (optarg);
        printf ("Using card %d device %d \n", card, dev);
        break;
       default:
        return 1;
       }
      }


      if ((rtn = snd_ctl_open (&ctl_handle, card)) < 0)
      {
       fprintf (stderr, "snd_ctlr_open failed: %s\n", snd_strerror (rtn));
       return -1;
      }

      if ((rtn = snd_mixer_open (&mixer_handle, card, dev)) < 0)
      {
       fprintf (stderr, "snd_mixer_open failed: %s\n", snd_strerror (rtn));
       snd_ctl_close (ctl_handle);
       return -1;
      }


      if (optind >= argc)
       display_groups (mixer_handle, argc - optind, argv + optind);
      else if (stricmp (argv[optind], "groups") == 0)
       display_groups (mixer_handle, argc - optind, argv + optind);
      else if (stricmp (argv[optind], "group") == 0)
```

```
  modify_group (mixer_handle, argc - optind, argv + optind);
 else if (stricmp (argv[optind], "switches") == 0)
  display_switches (ctl_handle, dev, argc - optind, argv + optind);
 else if (stricmp (argv[optind], "switch") == 0)
  modify_switch (ctl_handle, dev, argc - optind, argv + optind);
 else
  fprintf (stderr, "Unknown command specified \n");
 snd_mixer_close (mixer_handle);
 snd_ctl_close (ctl_handle);
 return (0);
}
```

Appendix D
ALSA and `libasound.so`

The only supported interface to the ALSA 5 drivers is through `libasound.so`. Direct use of *ioctl()* commands isn't supported because of the requirements of the ALSA API. It uses *ioctl()* commands in ways that are illegal in the QNX Neutrino RTOS (e.g., passing a structure that contains a pointer through an *ioctl()*).

The `asound` library is licensed under the **Library** GNU Public License (LGPL).

We include the `asound` library only as a shared library (`libasound.so`), and not as a static library. We intend to gradually improve the quality and number of services that this library provides; by linking against shared libraries, you'll receive the benefits of improvements without recompiling.

Appendix E
What's New in This Release?

This appendix describes the changes made in each release.

What's new in QNX Neutrino 6.6

snd_pcm_capture_go() **(p. 147)**

Start a PCM capture channel running

snd_pcm_capture_pause() **(p. 149)**

Pause a channel that's capturing

snd_pcm_capture_resume() **(p. 153)**

Resume a channel that was paused while capturing

snd_pcm_channel_go() **(p. 157)**

Start a PCM channel running

`snd_pcm_channel_params_t` **(p. 167)**

This structure now includes a *frags_buffered_max* member. If this is set, `io-audio` may block the caller after fewer than *frags_max* fragments have been passed, if it chooses, but won't block the client before *frags_buffered_max* fragments have been written.

snd_pcm_channel_pause() **(p. 171)**

Pause a channel

snd_pcm_channel_resume() **(p. 175)**

Resume a channel that was paused

`snd_pcm_channel_status_t` **(p. 184)**

The following members have been added to the structure:

- *status_data*
- *stop_time*
- *hw_device*

snd_pcm_find() **(p. 192)**

We've corrected the values of the *mode* argument.

snd_pcm_get_audioman_handle() **(p. 209)**

Retrieve an audioman handle that's bound to a PCM stream *snd_pcm_link()* **(p. 218)**

Link two PCM streams together

snd_pcm_open_name() **(p. 223)**

To enable echo cancellation and noise reduction, specify a name of `voice`.

snd_pcm_open_preferred() **(p. 226)**

We've described the format of the `preferences` file.

snd_pcm_playback_drain() **(p. 229)**

This function actually returns -`EINVAL` if the PCM device state isn't ready.

snd_pcm_playback_go() **(p. 233)**

Start a PCM playback channel running

snd_pcm_playback_pause() **(p. 235)**

Pause a channel that's playing back

snd_pcm_playback_resume() **(p. 239)**

Resume a channel that was paused while playing back

snd_pcm_plugin_playback_drain() **(p. 249)**

This function actually returns -`EINVAL` if the PCM device state isn't ready.

snd_pcm_plugin_set_enable() **(p. 258)**

Enable plugins that have been disabled

snd_pcm_set_audioman_handle() **(p. 279)**

Bind an audioman handle to a PCM stream

snd_pcm_unlink() **(p. 281)**

Detach a PCM stream from a link group

What's new in QNX Neutrino 6.5.0 Service Pack 1

snd_pcm_plugin_set_src_method() **(p. 260)**

Set the system's source filter method (plugin-aware)

What's new in QNX Neutrino 6.5.0

Voice conversion

The `libasound` library now supports devices that have more than two channels, and it provides a mechanism that lets you configure how the voice converter plugin replicates or reduces the voices or channels. For more information, see "*Controlling voice conversion* (p. 28)" in the Playing and Capturing Audio Data chapter.

`snd_pcm_channel_params_t` (p. 167)

This structure now includes a *sw_mixer_subchn_name* member that you can use to assign a name to the software mixer subchannel.

***snd_pcm_plugin_get_voice_conversion()* (p. 243)**

Get the current voice conversion structure for a channel

***snd_pcm_plugin_read()* (p. 253), *snd_pcm_plugin_write()* (p. 274), *snd_pcm_read()* (p. 277)**

These functions indicate an error of `EIO` if the channel isn't in the prepared or running state.

***snd_pcm_plugin_set_voice_conversion()* (p. 264)**

Set the current voice conversion structure for a channel

`snd_pcm_voice_conversion_t` (p. 282)

Data structure that controls voice conversion

`wave.c`*, *`waverec.c`*, *`mix_ctl.c`

We've updated these examples.

***snd_pcm_plugin_update_src()* (p. 272)**

Get the size of the next fragment to write

***snd_pcm_plugin_src_max_frag()* (p. 268)**

Get the maximum possible fragment size

***snd_pcm_plugin_set_src_mode()* (p. 262)**

Set the system's source mode

What's new in QNX Neutrino 6.4

snd_mixer_open_name() **(p. 126)**

Create a connection and handle to a mixer device specified by name

snd_pcm_open_name() **(p. 223)**

Create a handle and open a connection to an audio interface specified by name

What's new in QNX Neutrino 6.3

snd_ctl_mixer_switch_list() **(p. 73)**

Get the number and names of control switches for the mixer

snd_ctl_mixer_switch_read() **(p. 75)**

Get a mixer switch setting

snd_ctl_mixer_switch_write() **(p. 77)**

Adjust a mixer switch setting

`snd_switch_t` **(p. 287)**

Information about a mixer's switch

`mix_ctl.c`

A sample application that captures the groups and switches in the mixer

What's new in QNX Neutrino 6.2

The QNX Sound Architecture has evolved away from ALSA. You should reread this entire guide.

What's new in QNX Neutrino 6.1

snd_pcm_channel_info() **(p. 159)**

Removed the SND_PCM_CHNINFO_BATCH flag because it was deprecated in the source code.

Glossary

ADC

Analog Digital Converter. This converts an analog audio signal into a digital stream of samples.

ALSA

Advanced Linux Sound Architecture.

capture group

A mixer group that contains up to one volume, one mute, and one input selection element.

codec

Compression-Decompression module or Coder-Decoder.

DAC

Digital Analog Converter. This converts a digital stream of samples into an analog signal.

element

See *mixer element*.

group

See *mixer group*.

MIC

Microphone.

mixer element

A component of an audio mixer, with a single, discrete function.

mixer group

A collection or group of elements and associated control capabilities.

PCI

Peripheral Component Interconnect (personal computer bus).

PCM

Pulse Code Modulation. A technique for converting analog signals to a digital representation.

playback group

A mixer group that contains up to one volume element and one mute element.

QSA

QNX Sound Architecture.

SRC

Sample Rate Conversion.

subchannel

The collection of resources that a single connection to a client uses within a PCM device (playback or capture).

Index

F

H

I

L

M

N

O

P