QNX® Software Development Platform 7.0

QNX Neutrino Audio Developer's Guide

BlackBerry | QNX

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: March 06, 2020

Contents

About This Guide

The *Audio Developer's Guide* is intended for developers who want 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*
Playing and recording sound	*Playing and Capturing Audio Data*
The structure of a mixer	*Mixer Architecture*
Some tips for reducing audio latency	*Optimizing Audio*
Audio library functions	*Audio Library*
How to code an AFM control utility in C	*afm_ctl.c example*
How to code an APX control utility in C	*apx_ctl.c example*
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 mixer control utility in C	*mix_ctl.c example*
Why **libasound.a** isn't offered	*ALSA and libasound.so*
How to use audio concurrency management functionality	*Audio Concurrency Management*
Terms used in this guide	*Glossary*

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.

<sys/asound.h>, <sys/asoundlib.h>

Header files:

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`
Constants	NULL
Data types	`unsigned short`
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.

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

DANGER: 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. QSA also provides audio concurrency management that can manage audio ducking, audio preemption, and volume ramping.

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.

QSA with QNX Acoustics Management Platform

QNX Acoustics Management Platform (AMP) enables an audio pathway from capture to playback that is optimized for low-latency, high frame-rate, small frame size, real-time acoustic signal processing applications.

QNX AMP extends QSA with signal processing functionality using the `io-audio` Low-Latency Audio Architecture, into which one or more AMP Functional Modules (AFMs) and Acoustic Processing Extension libraries (APXs) are loaded.

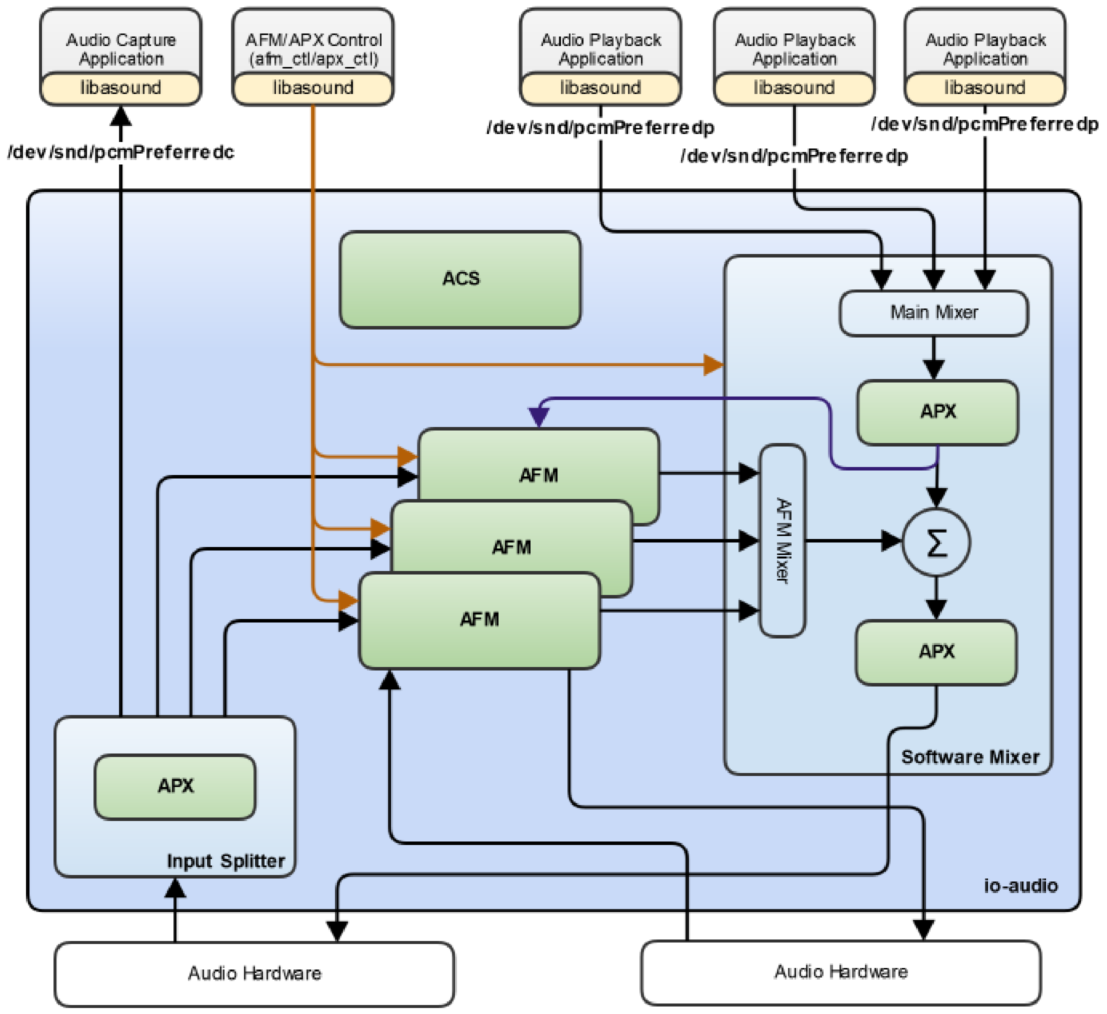

Figure 1: AMP audio flow.

AFMs are audio processing modules that connect to PCM devices in the digital domain to perform specialized audio processing functions. In most cases, an AFM is configured to source data from a PCM input splitter and sink data to a PCM software mixer. This configuration allows the AFM to run in parallel with other audio functions and applications that use the same audio PCM hardware device. AFMs provide solutions for features such as acoustic echo cancellation, chime and safety alert monitoring, engine order reduction, engine sound enhancement, external pedestrian alerts, in-car communication, and noise reduction. An AFM can also act as a simple bridge between capture on one PCM device and playback on another.

In contrast, APXs reside within PCM software mixers and input splitters, where they expand and optimize the perceptual quality of an audio stream, as well as provide tap points for audio streaming and injection for analysis purposes.

The low-latency software framework enables the implementation of demanding acoustic algorithms on application processors rather than dedicated Digital Signal Processors (DSPs). It extends the existing audio infrastructure in QNX Neutrino in a modular, completely backwards-compatible way so that applications such as telephony, speech recognition, navigation, and media playback all run entirely unmodified and interact positively with the acoustic signal processing functions of the system.

Concurrent access on playback and capture paths

QSA allows a single audio input and output hardware controller to service one or more microphones and one or more speakers within your target environment. To be able to share these devices between modules/processes/threads concurrently, a PCM software mixer on the playback path mixes multiple audio streams into a single output audio stream that targets the main speakers, and a PCM input splitter on the capture path replicates and distributes the single microphone (mono or multichannel) audio stream to different modules/processes/threads. An example of concurrent access on the playback path is having the media player, the In Car Communications (ICC) module, and the QNX Acoustics for Voice (QAV) module concurrently writing audio to the playback device for one or more cabin speakers. An example on the capture path is having a voice recognition application, the ICC module, and the QAV module concurrently sourcing data from the capture device for one or more cabin microphones. Without the PCM software mixer for playback and the PCM input splitter for capture, only a single module/process/thread can access a hardware device at any given time.

LiveAMP and Acoustics Control Server (ACS)

QNX AMP includes LiveAMP, a set of tuning, diagnosis, and analysis tools that enables real-time adjustment of system parameters, signal streaming, and injection at multiple tap-points in the system, as well as real-time spectrum and waveform displays. AFMs are controlled remotely using the Acoustics Control Server (ACS), which seamlessly links LiveAMP to the AFM and APX modules in `io-audio`.

AMP audio devices

In addition to the audio devices supported by QSA, AMP publishes the following devices:

AFM devices (/dev/snd/afmC*x*D*y*)

AFM devices are the entry points for control and status of an AFM. The *snd_afm_* ()* functions can be used to open and interact with an AFM. To simplify AFM access, a symbolic link that corresponds to the related acoustic processing library is also published (e.g., for the In-Car Communication module, **/dev/snd/icc**).

AFM PCM devices (/dev/snd/pcmC*x*D*y*)

Some AFMs create a PCM device to provide an application with an audio interface to AFM processing. This device simply appears as a new playback, capture, or playback and capture PCM device with a card and device number that matches the associated AFM. Data written to this class of PCM playback device passes through the AFM processing algorithms before being passed to a PCM hardware or software mixer device. The *snd_pcm_*()* functions can be used to open and interact with the PCM device in the same manner as non-AFM PCM devices. To simplify PCM access, symbolic links (ending with "p" for playback or "c" for capture) are also published (e.g., for the QNX Acoustics for Voice module, **/dev/snd/voicep** and **dev/snd/voicec** are added).

AFM mixer devices (/dev/snd/mixerC*x*D*y*)

AFMs that create a PCM device can also create a mixer device if volume control is enabled. The card and device number for the mixer device match the associated AFM. The *snd_mixer_* ()* functions can be used to open and interact with the mixer device in the same manner as non-AFM mixer devices.

Because Acoustic Processing Extension (APX) libraries reside within PCM software mixers or input splitters, they do not exist as separate devices. However, the *snd_apx_*()* functions can still be used to interact with them through the associated PCM device.

Audio library

AMP is supported by the *snd_afm_*()* and *snd_apx_*()* functions. In addition to basic controls and status, they also provide an interface to input Vehicle Input (VIN) data from the CAN bus for use by AFMs. For more information, see *Audio Library*.

Configuring io-audio

The `-c` option for `io-audio` allows you to configure the low-latency software framework using a configuration file that is loaded at startup.

You use the configuration file to load the DLLs that provide QNX AMP functionality.

Section	DLLs the section loads	DLL purpose
`[ctrl]`	`deva-ctrl-*` `deva-apx-*`	Create PCM and mixer device entry points and enable APX features
`[afm]`	`deva-afm-*`	Create AFM, PCM and mixer device entry points
`[acs]`	`deva-acs-link`	Create an AFM entry point and instantiate the ACS link

Audio devices are indexed based on the order in which their respective audio drivers are mounted in the `io-audio` process. Keep this in mind when you create your audio configuration file. The order of the `[ctrl]` sections determines the order in which the audio drivers are mounted into the `io-audio` process and thus determine the index of each audio device. Generally, this means you want the `[ctrl]` section for your main audio device (main microphone and speakers) to be the first `[ctrl]` section in the audio configuration file and any secondary audio devices (Bluetooth, radio tuner, and so on) and AFMs connected to those devices to follow it.

Audio device indexing uses card and device numbers. Each `[ctrl]` or `[afm]` creates a card, with the card indices starting at 0 and incrementing for each section of `[ctrl]` or `[afm]` type. Each card has 0 or many devices under it. (If there is an `[audiomgmt]` section, it is assigned card 0 unless a different value is specified using the `unit` key.)

You can use the `[acs]` section in the audio configuration file to load the `deva-acs-link` DLL. This DLL creates an afm entry point and instantiates the ACS link.

See `io-audio` in the *Utilities Reference* for a detailed description of io-audio configuration file.

AMP and audio concurrency management

The audio ducking feature affects AFM audio streams differently than other audio streams when routed through the AFM mixer path in the PCM software mixer (the default routing).

- QSA audio concurrency management never ducks the audio stream of an AFM.
- ASD, CSA, and ICC AFMs do not duck any other audio streams (in typical configurations, these AFMs are always on and would permanently duck other audio in the cabin, if it were allowed).
- You can configure the QAV and Bridge AFMs to duck specific audio streams that play through the main mixer audio path (for example, media playback). However, routing the Bridge AFM through the AFM mixer path is not a typical use case for that AFM.

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.

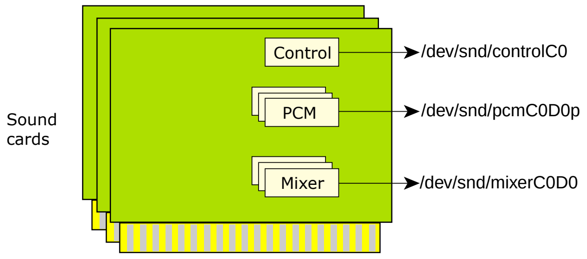

Figure 2: Cards and devices.

The devices include:

- *Control*
- *Mixer*
- *Pulse Code Modulation (PCM)*

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.

PCM devices are commonly created in playback and capture pairs creating two entries of the same device index under the same card with a direction identifier suffix ('p' for playback and 'c' for capture).

Each `deva-ctrl-*` DLL creates a logical sound card. For each card, there's a control (controlCx), as well as one or more mixers (mixerCxDy) and PCM devices(pcmCxDyz) entries. The card number (*x*) starts at zero and increments in the order that the DLLs are loaded. The driver (`unit=`*number*) option for the `io-audio` command can be used to specify the specific card number to be used for the DLL that follows it on the command line. See the *Utilities Reference* for more information.

You can list the devices that are on your system using the `ls` command. For example:

```
ls /dev/snd
```

For example:

```
# ls -l /dev/snd
total 0
lrw-rw-rw- 1 root root 0 May 31 11:11 capture -> pcmC0D0c
-rw-rw-rw- 1 root root 0 May 31 11:11 controlC0
-rw-rw-rw- 1 root root 0 May 31 11:11 mixerC0D0
-rw-rw-rw- 1 root root 0 May 31 11:11 pcmC0D0c
-rw-rw-rw- 1 root root 0 May 31 11:11 pcmC0D0p
lrw-rw-rw- 1 root root 0 May 31 11:11 pcmPreferredc -> pcmC0D0c
```

```
lrw-rw-rw- 1 root root 0 May 31 11:11 pcmPreferredp -> pcmC0D0p
lrw-rw-rw- 1 root root 0 May 31 11:11 playback -> pcmC0D0p0p
```

where:

- *C* represents the card
- *D* represents the device
- *p* represents playback
- *c* represents capture

In this case, the entry `pcmC0D0p` refers to card 0, device 0 for playback.

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.

Control events

You may have an application that needs to monitor audio concurrency management events and be notified when an event occurs.

These events track active audio streams and let you know what audio types are active on the system. You have to call *snd_ctl_read()* after the *select()* to read the event from the queue.

If it's a SND_CTL_READ_AUDIOMGMT_STATUS_CHG event, use *snd_ctl_ducking_status_read()* to get a list of all the active audio subchannels and information about their priority and ducking state.

Audio streams that have been user-paused (e.g., *snd_pcm_*_pause()*) are considered active.

For an example of how to monitor events and determine the state of active audio streams on the system, see the "*audiomgmt_monitor.c example*" appendix.

You use the `audiomgmt_id` key in the `[AUDIOMGMT]` section of the audio configuration file to specify the ducking output that you want to monitor (the audio concurrency management context or output) . For more information, see `io-audio` in the *Utilities Reference*.

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* 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.

PCM devices are commonly created in playback and capture pairs creating two entries of the same device index under the same card with a direction identifier suffix (“p” for playback and “c” for capture). 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*,” below.
- the data rates that the device can run at (48 KHz, 44.1 kHz etc.)
- the number of channels or voices 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 64 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.

Data format constants

The **<sys/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()* 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()*.

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

- *snd_pcm_format_big_endian()*
- *snd_pcm_format_linear()*
- *snd_pcm_format_little_endian()*
- *snd_pcm_format_signed()*
- *snd_pcm_format_unsigned()*

24-bit audio format

Data for 24-bit audio is represented as either 3 bytes or 4 bytes.

In most cases, audio files store 24-bit data as 3 bytes.

Because your hardware likely requires 32-bit aligned data access, the 24-bit data is usually converted to 4 bytes. The 24-bit sample in the 32-bit integer can have one of the following types of alignment:

- Left-aligned — QSA recognizes and handles a left-aligned 24-bit value in a 32-bit integer as a 32-bit sample value, with no loss in efficiency.
- Right-aligned — If the hardware requires 24-bit samples to be right-aligned in a 32-bit container, use the 24_4 data format constant. For playback, QSA upsamples 24-bit data to 32-bit for all processing and then converts it from 32-bit to 24_4 before passing it to the hardware. Likewise, for capture, QSA can accept 24_4 and convert it to 32-bit for processing.

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 is paused and is no longer playing the audio stream. To resume playing, you must explicitly call the appropriate *snd_pcm_*_resume()* function.

This state occurs when a user explicitly calls a *snd_pcm_*_pause()* function. It can also occur because the audio concurrency management policy in place has moved a subchannel that was previously in the SND_PCM_STATUS_SUSPEND to the SND_PCM_STATUS_PAUSED state.

SND_PCM_STATUS_SUSPENDED

Audio concurrency management has suspended the play of the audio stream based on the current audio concurrency management policies that are configured on the system.

SND_PCM_STATUS_SUSPENDED is a *transient* or temporary state that's controlled by the audio concurrency management policies running on the system. This transient state has two modes called *soft suspended* and *hard suspended*. For more information about these modes and how they work, see the "*Understanding preemption*" section in the *Audio Concurrency Management* chapter of this guide.

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 the audio stream must be reconfigured.

When the subchannel is put into the SND_PCM_STATUS_CHANGE state, the client must reconfigure the audio stream as the hardware capabilities have changed. For example, for an HDMI connection, the audio mode of the HDMI source (display, amplifier, etc.) may have changed from stereo to 5.1 surround sound, therefore, your subchannel must be reconfigured to enable the voice conversion plugin.

To reconfigure an audio stream, you must call *snd_pcm_plugin_params()* or *snd_pcm_channel_params()* and then the corresponding *snd_pcm_plugin_setup()* or *snd_pcm_channel_setup()* functions. If you want to check what new hardware capabilities are available, you can call *snd_pcm_channel_info()*, which gets information directly from the hardware.

After you reconfigure the audio stream, then you can call *snd_pcm_plugin_prepare()* or *snd_pcm_channel_prepare()* to move to the SND_PCM_STATUS_PREPARED state and to continue writing audio data to the subchannel.

The call to the *snd_pcm_plugin_params()* or *snd_pcm_channel_params()* functions may cause the fragment size to change. For that reason, ensure that after you make a *snd_pcm_*_params()* call, that you call the *snd_pcm_*_setup()* and *snd_pcm_*_prepare()* functions before you write your audio data.

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. This state occurs only if there aren't enough resources available on the system. 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.

Don't confuse the SND_PCM_STATUS_PREEMPTED state with the SND_PCM_STATUS_SUSPENDED state. When a subchannel gets preempted because audio concurrency management has occurred, it moves to SND_PCM_STATUS_SUSPENDED state.

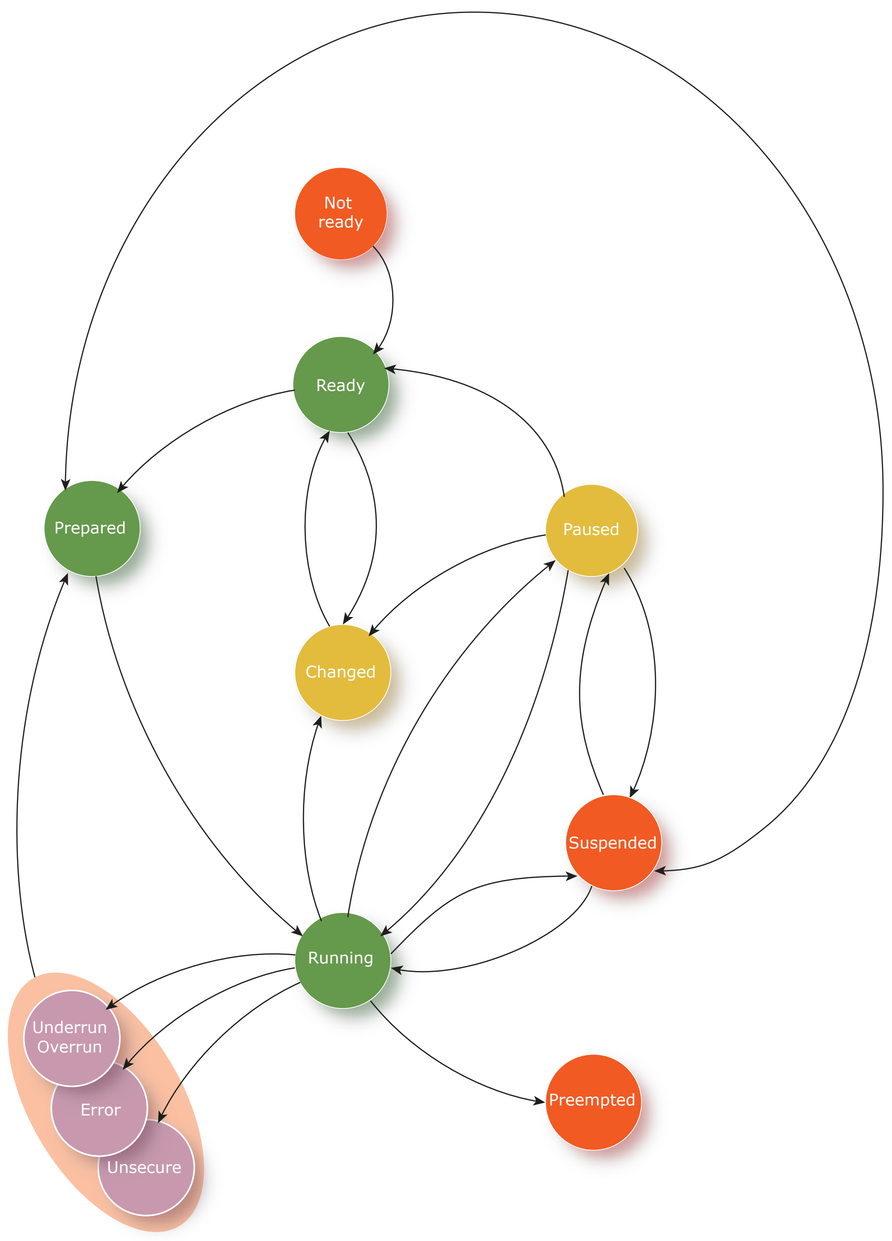

Figure 3: General state diagram for PCM devices.

In the figure above, the oval groups the Underrun/Overrun, Error, and Unsecure states together to indicate that each state can transition to the Prepared state.

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* chapter.

PCM software mixer

The PCM software mixer allows multiple application streams to play to a single hardware device concurrently by mixing together the audio samples of all of the application streams. When you enable

PCM software mixer reference streams in the audio configuration file (using `sw_mixer_max_references`), it also creates a capture device that provides the mixed playback audio stream as a reference audio stream/signal.

A PCM software mixer instance is automatically created for each hardware PCM playback device, even if the hardware supports multiple hardware subchannels. Requests from client applications to the hardware device are routed to the PCM software mixer automatically. You can disable the PCM sofware mixer using the `-o disable_sw_mixer` option.

When the PCM software mixer is not disabled, the CPU is used more than if you use the hardware device without the mixer, even when there is only one stream.

The PCM software mixer does not use a separate PCM device. Instead, it directly overlays the hardware PCM device created by the `deva-ctrl-*` DLL. This overlay means you can't access the hardware directly.

When you enable PCM software mixer media references, a new PCM capture device is created with an index that is one greater than the software mixer. When you capture from this device, you get the mixed audio stream from the application mixer stage. You cannot capture from the PCM software mixer reference if the PCM software mixer isn't actively mixing audio streams.

Client application fragment size

Client applications that attach to a PCM software mixer device can use a fragment size (in bytes) that is any multiple of the PCM software mixer's underlying fragment size. For example, if the PCM software mixer's fragment size is 4 KB, then the application's fragment can be a multiple of 4 KB.

You can use one of the following options to size the PCM software mixer fragments:

`sw_mixer_ms`

When you use this option, `io-audio` calculates a fragment size based on the requested period and the audio controller configuration. You can only specify this option in the audio configuration file. Use the following format:

```
sw_mixer_ms=<milliseconds per fragment>
```

<milliseconds per fragment> must be an integer.

`sw_mixer_samples`

Use this option to specify a fragment size using samples instead of a value in millseconds. Use the following format:

```
sw_mixer_samples=<samples per fragment>
```

Use the following formula to calculate *<samples per fragment>*:

Audio controller sample rate * Desired time period / Audio controller number of channels

For example, if the audio controller sample rate is 48 kHz, the time period is 16 ms, and the number of audio controller channels is two, *<samples per fragment>* is 768:

48000Hz * 16ms / 1000 = 768

You cannot use `sw_mixer_samples` if you are using the PCM input splitter.

Pausing the software mixer audio stream

When your subchannel goes through the software mixer ("sw_mixer") and you pause the audio stream (by calling *snd_pcm_*_pause()*), the *snd_pcm_*_pause()* call blocks until the software mixer completes ramping down the volume to zero. The subchannel transitions to the PAUSED state only after the volume is zero. If you don't use a software mixer, no ramping occurs.

Avoiding distortion with a predictive limiter

The PCM software mixer operates on 16 or 32-bit PCM data. When mixing multiple audio streams, the accumulated audio data can overflow and may need to be clamped to a minimum/maximum of 16 or 32-bit values. This clamping causes distortion, which is more noticeable the more the signal has overflowed into an out-of-bounds region. The degree and frequency of the overflow increases with the number of audio streams that are concurrently mixed.

To avoid the distortion, you can enable a predictive limiter (`sw_mixer_limiter`) on the accumulated mixer output that uses a one millisecond delay to look ahead and observe the impending amplitude of the output audio signal. Using that information, it calculates an attenuation that reduces the signal below the PCM limit. If the output signal is below the limit, the attenuation is gradually reduced back to zero. This provides a much smoother signal response than direct clamping, and allows many audio streams to be mixed concurrently without any noticeable audible distortion.

The use of a limiter is recommended on target systems where multiple audio streams are played concurrently and where audio streams use the full 16 or 32-bit PCM range. If there's normally only one audio stream playing or the audio streams that are being mixed have low amplitude, then the limiter isn't required. It's important to note that the playback output incurs a one millisecond delay if the limiter is enabled. For more information, see the `sw_mixer_limiter` configuration options.

PCM input splitter

A PCM input splitter is a virtual, capture-only device that directly overlays a PCM capture device to enable support for multiple subchannels on hardware that would otherwise support only a single subchannel. The PCM input splitter supports up to 10 concurrently attached subchannels.

The PCM splitter allows multiple processes and threads to concurrently capture audio from the same hardware device. For example, you could use different processes to capture audio for both voice recognition and file recording from the same capture device. The PCM input splitter can be enabled for a PCM capture device via the audio configuration file using the `input_splitter_enable` key. By default, the PCM input splitter is disabled.

Both the PCM input splitter and PCM software mixer must operate at the same data fragment size, which you set via the `sw_mixer_ms` option. Because you cannot use a samples value to specify the fragment size the PCM input splitter uses, you cannot size the PCM software mixer fragments using samples (the `sw_mixer_samples` option) when you use the input splitter.

For information about using an input splitter, see description for the `input_splitter_enable` key in the “Audio configuration file” section of “`io-audio`” in the *QNX Neutrino Utilities Reference* guide.

PCM plugin converters

In some cases, an application has data in one format, 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.

Plugin converters and PCM plugin API functions

The format conversion is accomplished by invoking a series of *plugin converters*, each capable of doing a very specific job. For 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.

To minimize CPU usage, the minimum number of converters is invoked to translate the input format to the output format. 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* 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()*, *snd_pcm_channel_setup()*, and *snd_pcm_channel_status()*. These behave as *snd_pcm_plugin_params()*, *snd_pcm_plugin_setup()*, and *snd_pcm_plugin_status()*, 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.

Polyphase SRC plugin converter

Because the polyphase SRC plugin converter provides the best quality rate conversion with the least amount of CPU usage, QSA always tries to use it as the source filter method, regardless of the filter set using *snd_pcm_plugin_set_src_method()*.

It supports the following sample rate conversions:

- 8000 to 12000, 16000, 24000, 32000, or 48000
- 11025 to 8000, 12000, 22050, 24000, or 44100
- 12000 to 8000, 11050, 24000, or 48000
- 16000 to 8000, 32000, 24000, 48000, or 96000
- 22050 to 8000, 11025, 16000, 24000, 44100, 48000, or 88200
- 24000 to 8000, 12000, 16000, 22050, 48000, or 96000
- 32000 to 8000,16000, 48000, 96000, or 192000
- 44100 to 11020, 16000, 22050, 32000, 48000, 88200, 96000, or 1764000
- 48000 to 8000, 12000, 16000, 24000, 32000, 44100, 96000, or 192000
- 88200 to 22050, 32000, 44100, 96000, 192000, or 176400

- 96000 to 16000, 24000, 32000, 88200, or 192000
- 176400 to 44100, 88200, or 19200
- 192000 to 32000, 48000, 96000, or 176400

If the client requests a conversion that is not supported, QSA uses one of the other supported SRC methods (set using *snd_pcm_plugin_set_src_method()*), which support any sample rate conversion but may produce lower-quality conversions or use more CPU resources.

QSA also uses one of the other SRC methods if asynchronous SRC is enabled (set using *snd_pcm_plugin_set_src_mode()*).

PCM events

PCM events are sent when changes occur to a subchannel.

There many situations where it's useful to notify an application when an event occurs to the subchannel so that the application can take appropriate actions. For example, an application may maintain its own states, and so needs to update its application-specific states.

PCM events can be a state change or when the subchannel has been muted by the system. For example, events are when audio concurrency management moves a subchannel from the RUNNING state to the SUSPENDED state. It's important to note that PCM events aren't generated when API calls are made; for example, a call to `snd_pcm_*_pause()` won't generate an event.

By default, applications won't receive events unless they call the *snd_pcm_set_filter()* to register to receive them. To register for events, a bitmask must be applied to the `enable` member in the *`snd_pcm_filter_t`* argument to the *snd_pcm_set_filter()* as shown here:

```
  /* Enable PCM events */
 snd_pcm_filter_t pevent;
 pevent.enable = ( (1<<SND_PCM_EVENT_AUDIOMGMT_STATUS) |
                   (1<<SND_PCM_EVENT_AUDIOMGMT_MUTE) |
                   (1<<SND_PCM_EVENT_OUTPUTCLASS) );
 snd_pcm_set_filter(pcm_handle, SND_PCM_CHANNEL_PLAYBACK, &pevent);
```

You can use *snd_pcm_get_filter()* to see which events you registered for.

To work with the PCM events, call *snd_pcm_channel_read_event()*. That call is a non-blocking call where you can then use *select()* (with *exceptfds*) or call *poll()* (with POLLRDBAND) to retrieve the PCM events from your queue. For information about those functions, see the QNX Neutrino *C Library Reference*.

When you retrieve an event, use the `snd_pcm_event_t` structure to determine the event type you received. You can use the `data` member (a union) to get event data. These are the available event types:

SND_PCM_EVENT_AUDIOMGMT_STATUS

This event type indicates that an audio concurrency management related state change occurred. The data member of the event contains the previous state (`old_status`) and new state (`new_status`). Here are valid state changes that cause an event to be generated:

- From RUNNING (SND_PCM_STATUS_RUNNING) to SUSPENDED (SND_PCM_STATUS_SUSPENDED) or PAUSED (SND_PCM_STATUS_PAUSED)
- From SUSPENDED to RUNNING or PAUSED
- From PAUSED to SUSPENDED

If forced ducking is enabled and the subchannel isn't actively streaming, here are the state changes that cause an event to be generated:

- From PREPARED (SND_PCM_STATUS_PREPARED) to SUSPENDED (SND_PCM_STATUS_SUSPENDED)
- From SUSPENDED (SND_PCM_STATUS_SUSPENDED) to PREPARED (SND_PCM_STATUS_PREPARED)
- From SUSPENDED (SND_PCM_STATUS_SUSPENDED) to READY (SND_PCM_STATUS_READY)

SND_PCM_EVENT_AUDIOMGMT_MUTE

Mute events occur for audio types that have their volume ducked to zero due to audio concurrency management policies being applied on the system. Though the volume is zero, it's still in the RUNNING state. If preemption is configured, being ducked to zero won't generate this event, instead an SND_PCM_EVENT_AUDIOMGMT_STATUS event is generated.

SND_PCM_EVENT_OUTPUTCLASS

The event indicates a change to the output class of the PCM playback device. You can use the *old_output_class* and *new_output_class* members in the `snd_pcm_outputclass_event_t` member to get the information about the previous output class and current output class, respectively.

PCM thread priorities

The PCM software mixer and PCM input splitter threads run at a priority of either +1 from the highest audio client application priority or `data_thread_prio`, whichever is higher. To make sure that the threads closest to the hardware have the highest priority, the client application priorities should never be more than -2 from `intr_thread_priority`.

For example, if the priority of the client thread is 48, the PCM software mixer thread is 49 (48+1), which is one less than the default `intr_thread_priority` value of 50.

For more information on `data_thread_prio` and `intr_thread_priority`, see the entry for `io_audio` in the *Utilities Reference*.

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 play back 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()

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

snd_pcm_open()

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

snd_pcm_open_preferred()

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` structure, and then calling *snd_pcm_channel_params()* or *snd_pcm_plugin_params()*. The difference between the functions is that the second one uses the plugin converters (see "*PCM plugin converters*" 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()

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()

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` 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, then an audio subchannel is acquired based on the provided parameters. The QNX Sound Architecture (QSA), on a best effort basis, uses the fragment size *frag_size* as requested, but the actual fragment size that's used may differ based on hardware alignment restrictions and potential data conversions that may exist. In the end, you must use the fragment size that's returned from *snd_pcm_plugin_setup()*. For more information, see the note about fragment sizes in the *Optimizing Audio* chapter of this guide.

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()* or *snd_pcm_plugin_setup()*, 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` structure that they fill with information about how the channel is configured, including the true *frag_size*.

Controlling voice conversion

Configuration of the **libasound** library is based on the maximum number of voices supported in hardware.

The *snd_pcm_plugin_params()* function instantiates a voice converter in the following scenarios:

- The source and destination voices do not use the same channel mappings.
- The number of source and destination voices is different.

If the number of source voices and their mapping matches the destination, the converter isn't invoked.

Default channel mappings

The following channel mappings are used for application playback streams. If the destination uses different mappings, the voice converter is invoked to map the source channels to appropriate destination channels.

Number of channels	Default mapping
1	Front left OR mono
2	• 1 — Front left • 2 — Front right
4	• 1 — Front left • 2 — Front right • 3 — Rear left • 4 — Rear right
6	• 1 — Front left • 2 — Front right • 3 — Front center • 4 — Low-frequency effects

Number of channels	Default mapping
	• 5 — Rear left • 6 — Rear right
8	• 1 — Front left • 2 — Front right • 3 — Front center • 4 — Low-frequency effects • 5 — Rear left • 6 — Rear right • 7 — Surround left • 8 — Surround right

If the stream has a number of voices other than the ones above, the voice converter does not perform any remapping by default. The application must query the driver channel map (chmap) and configure the voice matrix to do any required mapping of the channels. See "*Overriding the default voice conversion*".

Destination has more voices (upmixing)

When there are a larger number of voices on the destination, the default voice conversion behavior is as follows:

From	To	Conversion
Mono	Stereo	Replicate channel 1 (left) to channel 2 (right).
Mono	4-channel	Replicate channel 1 to the front left, front right, rear left, and rear right. If the hardware's channel map has only some of these mappings, channel 1 is replicated on those channels and the remaining ones are silent. If the hardware's channel map has none of these mappings, map channel 1 on the first available channel and the remaining channels are silent.
Stereo	4-channel	Replicate channel 1 to front left and rear left and channel 2 to front right and rear right. If the hardware's channel map has only some of these mappings, replication happens for the available mappings only. Any remaining channels are silent.

From	To	Conversion
		If the hardware's channel map has none of these mappings, map channel 1 to the first available channel and channel 2 to the next available one. The remaining channels are silent.
Mono	More than 4 channels	Replicate channel 1 to the front left, front right, rear left, and rear right. All remaining channels are silent. If the hardware's channel map has only some of these mappings, channel 1 is replicated on those channels and the remaining ones are silent. If the hardware's channel map has none of these mappings, map channel 1 on the first available channel. The remaining channels are silent.
Stereo	More than 4 channels	Replicate channel 1 to front left and rear left and channel 2 to front right and rear right. All remaining channels are silent. If the hardware's channel map has only some of these mappings, replication happens for the available mappings only. Any remaining channels are silent. If the hardware's channel map has none of these mappings, map channel 1 to the first available channel and channel 2 to the next available one. All remaining channels are silent.

Destination has fewer voices (downmixing)

When there are fewer voices on the destination, the default voice conversion behavior is as follows:

From	To	Conversion
Stereo	Mono	Average channels 1 and 2.
4-channel	Mono	Average front-left, front-right, rear-left, and rear-right channels (or whichever of these are present). Any other mappings are ignored.
4-channel	Stereo	Average front left and rear left to front left, and front right and rear right to front right. Any other mappings are ignored. If no mappings are available, the first two channels of the source are mapped to the first two channels of the destination. The remaining two source channels are ignored.
More than 4 channels	Mono	Average front-left, front-right, rear-left, and rear-right channels (or whichever of these are present). All other mappings are ignored. If no mappings are available, the first channel of the source is mapped to the destination channel and all other source channels are ignored.
More than 4 channels	Stereo	Average front left and rear left to front left, and front right and rear right to front right. All other mappings are ignored.

From	To	Conversion
		If no mappings are available, first two channels of the source are mapped to the two destination channels and the additional source channels are ignored.
Any number greater than the destination	Other than stereo or mono (e.g., 5.1 surround sound (6 channels))	The first channel of the source is mapped to the first destination channel, the second source channel to the second destination channel, and so on, until all the destination channels are mapped to. The remaining source channels are ignored.

In QNX Neutrino 6.5.0, **libasound** converted stereo to mono by simply dropping the right channel. In QNX Neutrino 6.6.0 or later, **libasound** averages the left and right channels to generate the mono stream.

Overriding the default voice conversion

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

snd_pcm_plugin_get_voice_conversion()

Get the current voice conversion structure for a channel

snd_pcm_plugin_set_voice_conversion()

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 low-frequency effects (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 you call *snd_pcm_plugin_get_voice_conversion()* before the voice conversion plugin has been instantiated, the function fails and returns -ENOENT. In QNX Neutrino 6.6 or later, *snd_pcm_plugin_set_voice_conversion()* instantiates the plugin if it doesn't already exist.

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()* if you're using the plugin interface
- *snd_pcm_channel_prepare()*, *snd_pcm_capture_prepare()*, or *snd_pcm_playback_prepare()* 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*" and "*If the PCM subchannel stops during capture*," 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()* 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.

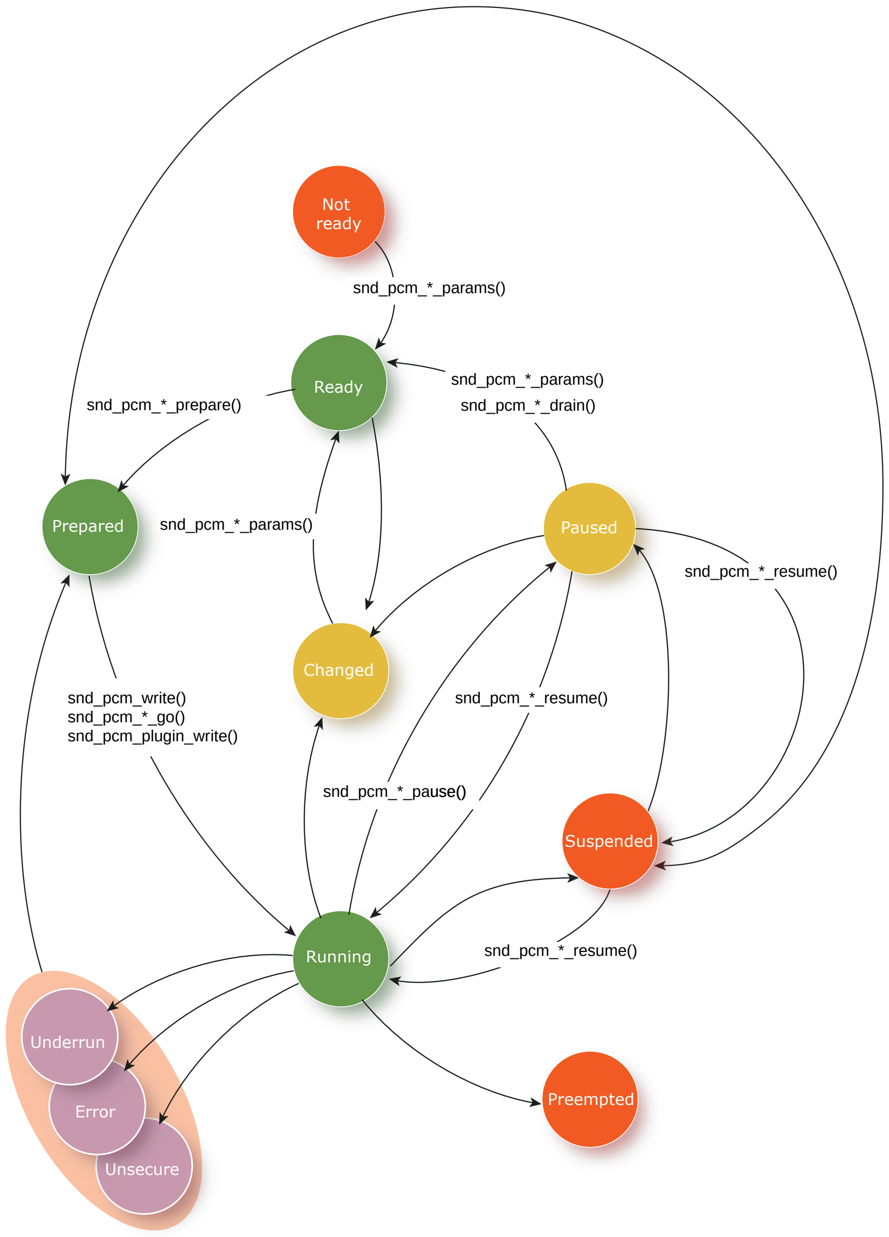

Figure 4: State diagram for PCM devices during playback.

In the figure above, the oval groups the Underrun, Error, and Unsecure states together to indicate that each state can transition to the Prepared state

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()* or *snd_pcm_plugin_params()*
READY	PREPARED	Calling *snd_pcm_channel_prepare()*, *snd_pcm_playback_prepare()*, or *snd_pcm_plugin_prepare()*
PREPARED	RUNNING	Calling *snd_pcm_playback_go()*, *snd_pcm_channel_go()*, *snd_pcm_write()* or *snd_pcm_plugin_write()*
PREPARED	SUSPENDED	When forced ducking is enabled via a call to *snd_pcm_channel_audio_ducking()*, the state change occurs when the audio concurrency management policy in place has ducked the audio stream that's currently in the PREPARED (SND_PCM_STATUS_PREPARED) state. If forced ducking isn't enabled, this state change doesn't occur.
RUNNING	PAUSED	Calling *snd_pcm_channel_pause()* or *snd_pcm_playback_pause()*
PAUSED	RUNNING	Calling *snd_pcm_channel_resume()* or *snd_pcm_playback_resume()*
PAUSED	SUSPENDED	While the subchannel was paused, a higher priority or same priority subchannel preempted the paused subchannel (volume was ducked to zero). For that reason, the stream moves to the suspended (SND_PCM_STATUS_SUSPENDED) state.
PAUSED	READY	One of the following: • While the subchannel was paused, if you call *snd_pcm_*_params()* or *snd_pcm_*_drain()*, the stream moves to the READY state. • While the subchannel was paused, if you call *snd_pcm_*_flush()*, the playback resumes, and then the stream moves to the READY state after the buffered data is fully consumed.
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	CHANGED	The stream parameters have changed
RUNNING	ERROR	A hardware error occurred
RUNNING	SUSPENDED	Audio concurrency management has suspended the subchannel because a higher or same priority subchannel has run.
SUSPENDED	RUNNING	Audio concurrency management has moved the previously suspended subchannel back to the running (SND_PCM_STATUS_RUNNING)

From	To	Cause
		state and the subchannel starts to play again. The application must start writing data, otherwise an underrun condition can occur. This transition may also occur if a subchannel is in a soft suspended (SND_PCM_STATUS_SUSPENDED) state and calls *snd_pcm_channel_resume()* or *snd_pcm_playback_resume()*. For more information, see "*Understanding preemption*" section in the Audio Concurrency Management chapter of this guide.
SUSPENDED	PAUSED	The state occurred because the audio concurrency management policy in place is configured to `pause`, which means when audio concurrency management clears the suspension, a transition was made to the PAUSED (SND_PCM_STATUS_PAUSED) state rather the RUNNING (SND_PCM_STATUS_RUNNING) state. If you were previously in the PAUSED state before the suspension occurs, after suspension is lifted, you return to the PAUSED state. While in the SUSPENDED state, if a `snd_pcm_*_pause()` call was made, you would transition to the PAUSED state when the suspension is lifted.
SUSPENDED	PREPARED	If forced ducking was enabled via a call to *snd_pcm_channel_audio_ducking()*, then audio concurrency management changes from the SUSPENDED state to the PREPARED (SND_PCM_STATUS_PREPARED) when the suspension condition has been lifted.
SUSPENDED	READY	While the subchannel was suspended, if you call *snd_pcm_*_drain()*, the stream moves to the READY state.
UNDERRUN, UNSECURE, or ERROR	PREPARED	Calling *snd_pcm_channel_prepare()*, *snd_pcm_playback_prepare()*, or *snd_pcm_plugin_prepare()*
RUNNING	PREEMPTED	Audio is blocked because another **libasound** session has initiated playback, but there aren't enough system resources available; the audio driver has determined that the new session has higher priority and as a result, this session has been preempted.

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

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

snd_pcm_plugin_write()

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()*.

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()*.

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()* and *snd_pcm_plugin_write()* 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*," 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()* if you're using the plugins
- *snd_pcm_playback_drain()* if you aren't

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

- *snd_pcm_plugin_flush()* if you're using the plugins
- *snd_pcm_channel_flush()* or *snd_pcm_playback_flush()* 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()* if you're using the plugin interface
- *snd_pcm_channel_status()* if you aren't

Both of these functions fill in a `snd_pcm_channel_status_t` 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. The mmap plugin is disabled by default, but if you have it enabled, you can call *snd_pcm_plugin_set_disable()* to disable the mmap plugin.

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*."

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*
- *Capture states*
- *Receiving data from the PCM subchannel*
- *If the PCM subchannel stops during capture*
- *Stopping the capture*
- *Synchronizing with the PCM subchannel*

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()*, for controlling the mixer so that the application can set this up directly; it's described in the *Mixer Architecture* 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.

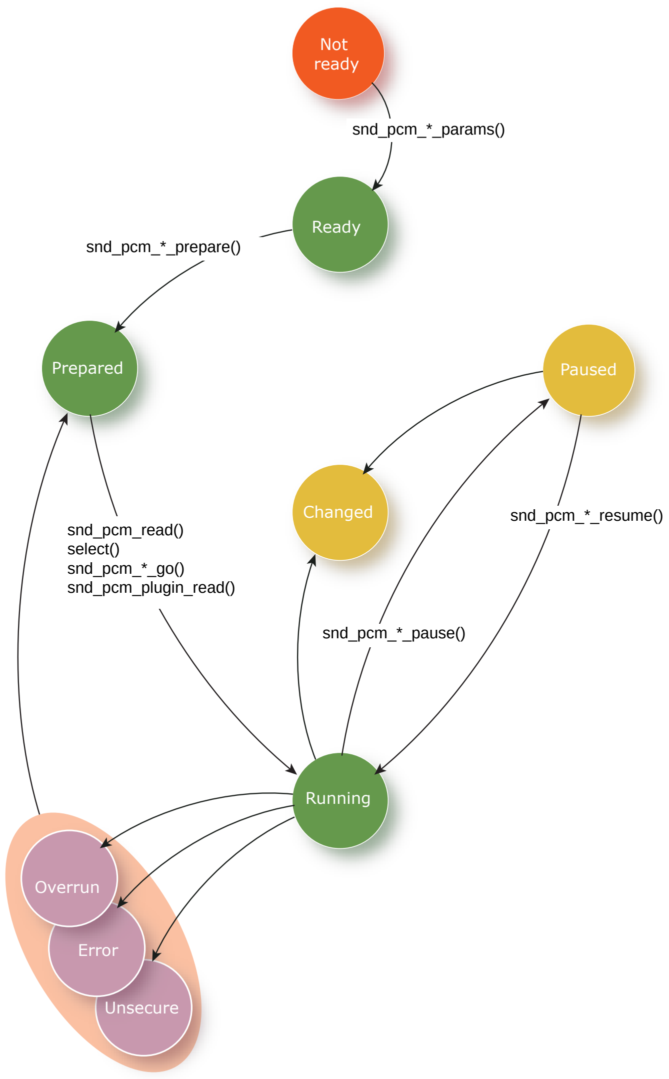

Figure 5: State diagram for PCM devices during capture.

In the figure above, the oval groups the Overrun, Error, and Unsecure states together to indicate that each state can transition to the Prepared state

The circle surrounding Overrun, Error, and Unsecure states indicates that each state can individually transition to the Prepared state. 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()* or *snd_pcm_plugin_params()*
READY	PREPARED	Calling *snd_pcm_capture_prepare()*, *snd_pcm_channel_prepare()*, or *snd_pcm_plugin_prepare()*

From	To	Cause
PREPARED	RUNNING	Calling *snd_pcm_capture_go()*, *snd_pcm_channel_go()*, *snd_pcm_read()*, or *snd_pcm_plugin_read()*, or calling *select()* against the capture file descriptors
RUNNING	PAUSED	Calling *snd_pcm_capture_pause()* or *snd_pcm_channel_pause()*
PAUSED	RUNNING	Calling *snd_pcm_capture_resume()* or *snd_pcm_channel_resume()*
RUNNING	OVERRUN	The hardware buffer became full during capture; *snd_pcm_read()* and *snd_pcm_plugin_read()* fail
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, or ERROR	PREPARED	Calling *snd_pcm_capture_prepare()*, *snd_pcm_channel_prepare()*, or *snd_pcm_plugin_prepare()*

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

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

snd_pcm_plugin_read()

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()* 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()*.

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()* and *snd_pcm_plugin_read()* fail) and the subchannel doesn't restart capturing.

The only ways to move out of this state are to close the subchannel or to re-prepare 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()* if you're using the plugins
- *snd_pcm_channel_flush()* or *snd_pcm_capture_flush()* 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()* if you're using the plugin interface

- *snd_pcm_channel_status()* if you aren't

Both of these functions fill in a `snd_pcm_channel_status_t` 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()*.

Chapter 3 Mixer Architecture

This section describes the mixer architecture.

You can usually build an audio mixer from a relatively small number of components. The software mixer is enabled by default, so anything that's directed to the PCM device is automatically redirected through the PCM software mixer. 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:

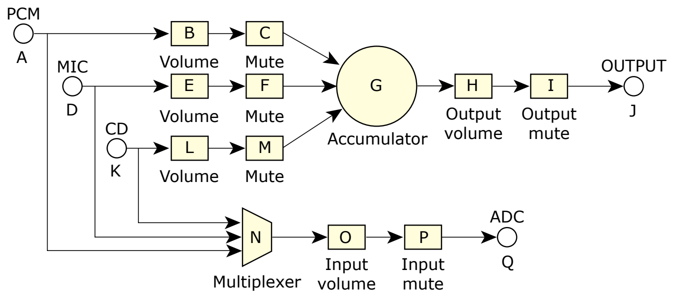

Figure 6: A simple sound card mixer.

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()* and *snd_mixer_element_write()* 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()* 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()* 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`; 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`) 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()*, 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()* or *snd_pcm_plugin_setup()*, 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()* 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* structure. The structure contains a pointer to where the groups' identifiers are to be stored (an array of *snd_mixer_gid_t* 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 (*snd_strerror()* 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()* API call to mask out events it's not interested in.

Applications use the *snd_mixer_read()* function to read the enqueued mixer events. The arguments to this function 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()*.

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()*.

Chapter 4
Audio Concurrency Management

You can manage multiple audio streams using various audio policies.

The policies control the following audio concurrency management features:

- *Audio ducking* (referred to as simply ducking) manages concurrent audio playback where playing one audio stream lowers the volume of another stream. Whether one audio stream causes another stream to duck (lower in volume) is determined by priorities that you configure.
- *Audio ramping* (referred to as simply ramping) gradually increases or decreases the volume of an audio stream so that it fades in or fades out. This effect makes ducking less abrupt.
- When *audio preemption* (or *preemption*) is configured and the system ducks an audio stream, it also temporarily suspends the playing of that stream. You specify preemption options by audio type.
- *Audio type volume controls* allow you to apply volume settings to all audio streams of a particular audio type.

You configure audio policies in the audio policy configuration file, which you specify using `policy_conf` in the `[AUDIOMGMT]` section of the audio configuration file. Each audio concurrency management context has its own audio policy configuration file and associated mixer device.

If your application needs to monitor the system's audio concurrency management changes, you can use control events. For more information, see "*Control events*" in the "Audio Architecture" chapter of this guide.

Understanding audio ducking

When *audio ducking* is enabled, all audio streams adhere to an audio ducking policy that's defined in the audio policy configuration file.

Each audio stream on the system should have a corresponding *audio type* defined in the audio policy configuration file. Calls to *snd_pcm_*_params()* with an audio type that doesn't match a defined audio type fail with EINVAL. If you call *snd_pcm_*_params()* and do not provide an audio type, then the `default` type is assigned. The first audio type that's defined at the top of the file has the highest priority and the last audio type defined in the file has the lowest priority. You can also configure audio types that have the same priority.

If multiple audio streams play concurrently, the higher-priority audio type listed in the audio policy configuration file applies ducking, which lowers the volume of the lower priority audio type. For example, if you receive a voice call while music is playing and the audio type of the call has a higher priority than the music audio type, the music is ducked so that the voice call is heard more clearly. The voice call is ducking the music and the music is ducked.

If audio streams are in the same PCM link group, audio concurrency management policies treat them as a single audio stream. Ducking and other policies are not applied between members of the same PCM link group.

Using audio ducking

To enable ducking, you create an *audio policy configuration file* that follows the rules defined in the "*Syntax of the audio policy configuration file*" section in this chapter.

Ducking is not available if you disable the software mixer (by specifying `disable_sw_mixer` option when you run `io-audio`). For more information, see the `disable_sw_mixer` option in the entry for `io_audio` in the *Utilities Reference*.

To enable ducking, you specify the location of audio policy configuration file using the `policy_conf` key in the `[AUDIOMGMT]` section of the audio configuration file.

For more information about the audio configuration file, see "Audio configuration file" in the entry for `io_audio` in the *Utilities Reference*.

Manually enabling ducking using APIs

Audio concurrency management evaluates ducking whenever an audio stream transitions into or out of the RUNNING state. However, if the stream is in the PREPARED state (call *snd_pcm_*_prepare()*), you can use the *snd_pcm_channel_audio_ducking()* function to make audio concurrency management apply ducking polices to an audio stream as if it was RUNNING, regardless of its current state.

You can call *snd_pcm_channel_audio_ducking()* to manually duck an audio stream only after a transition to the PREPARED state, but you can call it to cancel manual ducking at any time.

Manual ducking remains active until you use *snd_pcm_channel_audio_ducking()* to cancel it or reset the parameters using *snd_pcm_plugin_params()*.

Ducking and AFMs

Audio ducking works differently for AFMs depending on the configuration:

- If the audio configuration file routes the AFM playback audio through the main (media) path in the PCM software mixer value (i.e., `sw_mixer_route` is `main`), the AFM audio can duck and be ducked by other audio.
- If the audio configuration file routes the AFM playback audio through the AFM mixer path, it is never ducked (`sw_mixer_route` is `afm` by default). However, only certain AFMs, such as the Bridge and Voice AFMs, duck other audio streams. See "AMP and audio concurrency management" in "*QSA with QNX Acoustics Management Platform*".

Understanding the **`mix`** policy when using multiple audio types with the same priority

If you have multiple audio types with the same priority and `duck_same_prio_policy` set to `mix`, when those types of streams play together, their volume levels are ducked according to each type's `duck_same_prior_percent` setting.

Any new active streams immediately apply ducking to other audio streams with the same priority and the ducking settings of the other audio streams are applied to the new stream. If multiple ducking percentages apply, then the strictest (lowest) value is applied.

For audio streams with a lower priority than this kind of audio stream group (types with the same priority and configured as `mix`), the strictest (lowest) value of the active group members is also applied.

For example, three audio types (`multimedia1`, `multimedia2`, and `multimedia3`) are configured with same priority and for two channels (stereo). Each audio type has the following channel settings for `duck_same_prio_percent`:

- `multimedia1` is `ch0:10, ch1:50`
- `multimedia2` is `ch0:0, ch1:100`
- `multimedia3` is `ch0:100, ch1:100`

When audio streams that use two of these audio types play concurrently, the streams duck each other by the specified percentages.

However, when all three types play, the volume for each stream decreases to the lowest (strictest) value of the two other types and audio concurrency management ducks the audio streams to the following percentages:

- `multimedia1` to ch0=0 and ch1=100 (the lowest value between `multimedia2` and `multimedia3` is 10 for channel 0 and 100 for channel 1)
- `multimedia2` to ch0=10 and ch1=50 (the lowest value between `multimedia1` and `multimedia3` is 10 for channel 0 and 50 for channel 1)
- `multimedia3` to ch0=0 and ch1=50 (the lowest value between `multimedia1` and `multimedia2` is 10 for channel 0 and 50 for channel 1)

For a stream with a lower priority, if all these example streams are playing, the ducking percentages are ch0=0 and ch1=50 (the lowest values for each channel found in the group).

Here's what these example audio types look like in the audio policy configuration file:

```
...
...
[audio_type]
name=multimedia1
prio=same
duck_same_prio_policy=mix
duck_same_prio_percent=ch0:10,ch1:50
duck_lower_prio_percent=20

[audio_type]
name=multimedia2
prio=same
duck_same_prio_policy=mix
duck_same_prio_percent=ch0:0,ch1:100
duck_lower_prio_percent=20

[audio_type]
name=multimedia3
prio=same
duck_same_prio_policy=mix
duck_same_prio_percent=ch0:100,ch1:100
duck_lower_prio_percent=20
...
...
```

Understanding how multiple ducking levels cumulate

Higher priority audio types can each apply a ducking level to a lower priority audio type, which creates a cumulative ducking level.

This behavior ensures that the relative ducking levels are correct. For example, using the file shown in *"Example of an audio policy configuration file"*, a system plays three active audio streams that match the audio types `voice`, `ringtone`, and `multimedia`. Each stream starts with a volume level of `100`:

```
voice = 100
ringtone = 100
multimedia = 100
```

The volume decreases to the following levels after ducking occurs:

```
voice = 100
ringtone = 50
multimedia = 10
```

The `voice` audio type's `duck_lower_prio_percent` key causes both the lower priority audio types to adjust to 50% volume (50), but because the `ringtone` audio type's `duck_lower_prio_percent` key is set to 20, `ringtone` applies additional ducking to the lower priority audio type, `multimedia`. In this case, 20% of 50 is a volume level of 10.

Example of an audio policy configuration file

```
[audio_type]
name=voice
prio=same
duck_same_prio_policy=last_wins
duck_same_prio_percent=0
duck_lower_prio_percent=50

[audio_type]
name=ringtone
# priority of the audio type is decreased by one from previous audio type
prio=decr
# ducking policy of same priority is last_wins, i.e., the last added
# audio stream can duck previously  added audio streams of same priority
duck_same_prio_policy=last_wins
duck_same_prio_percent=50
duck_lower_prio_percent=20

[audio_type]
name=nav
prio=decr
duck_same_prio_policy=last_wins
# Channel 0 ducked to 50 percent, Channel 1 remains at 100 percent
# Ducking values default to 100 (no ducking) for channels not listed
duck_same_prio_percent=ch0:50,ch1:100
duck_lower_prio_percent=ch0:50,ch1:100

[audio_type]
name=tts_front
prio=decr
duck_same_prio_policy=last_wins
# Ducking values apply to all channels
duck_same_prio_percent=50
duck_lower_prio_percent=50

[audio_type]
name=tts_back
prio=same
duck_same_prio_policy=last_wins
duck_same_prio_percent=50
duck_lower_prio_percent=50

[audio_type]
name=multimedia
prio=decr
duck_same_prio_policy=last_wins
duck_same_prio_percent=0
```

```
duck_lower_prio_percent=50

[audio_type]
name=default
prio=decr
# Mix audio streams at same priority with the below ducking configuration
duck_same_prio_policy=mix
# 100 percent means that no ducking is performed
duck_same_prio_percent=100
duck_lower_prio_percent=noducking
```

Understanding volume ramping

Volume ramping (or simply ramping) allows you to control volume changes.

Using volume ramping alleviates artifacts, such as clicks or pop sounds, that typically occur when a change in volume is too great over a short period of time. You can also use volume ramping to gradually change the volume to get a “fade in” or “fade out” effect. For example, if you have a voice subchannel that ducks the music that's playing on another subchannel, you may want to use volume ramping so that when the voice call completes, rather than starting the audio so that it plays almost immediately at its original volume, you may want to gradually increase the volume.

You can use multiple segments for your *volume ramping* ramp so that you can build a curve for the ramp. If you choose to use one segment, the volume ramps is linear. Each volume segment defines two values, delimited by a colon; while each segment is delimited by a comma:

- A percentage of the duration specified (expressed as a percentage value between 1 and 100) based on a configured number of samples (duration of time).
- A percentage of the overall volume delta to increase the volume. This is a linear increase in volume for the segment. The volume delta is the difference between the current volume level and the volume level before the subchannel was muted or paused.

You can define multiple segments, and the percentages that provide for the time to increase the volume and the amount to increase the volume by must total 100. Or you can define one segment, which specifies a liner ramp in volume.

The ramp duration is the time in milliseconds for the entire ramp, or the total number of ramp segments. It is a configured value that you provide in the audio policy configuration file. For more information, see the “`[vol_ramp]`” subsection in the “*Syntax of the audio policy configuration file*” section of this chapter.

Volume ramping types

You can control the volume ramp for various audio concurrency management scenarios using the `[vol_ramp]` section in the audio policy configuration file. These are the scenarios that you can apply volume ramp to:

If you don't specify a volume ramp, the default for each of the scenarios is 20 milliseconds and the profile is a linear ramp (i.e., 100:100).

`ducking`

The volume ramp corresponds to any ducking-related volume changes made by audio concurrency management.

`pause_resume`

The volume ramp corresponds to pause and resume requests made using the `snd_pcm_*_pause()` or `snd_pcm_*_resume()` calls.

volume_mute

The volume ramp corresponds to volume or mute requests made using APIs. This value is used only if a duration isn't specified during a volume change request. For example, volume changes can be made by calling *snd_mixer_group_write()*, and the duration can be specified at that time.

Volume ramping example

The following shows you the configuration to use in the audio policy configuration file to use volume ramping.

Here's how the `[VOL_RAMP]` tag is used. The names that you can use are `volume_mute`, `pause_resume`, and `ducking`. The duration you set is in milliseconds. You can use the `profile` key-value pair to specify multiple sample percentages of the specified duration and volume increments (expressed as percentages) that can be used to curve the volume ramp; otherwise if you don't provide `profile`, the volume ramp is linear. For more information about the syntax, see "*Syntax of the audio policy configuration file*" in this chapter.

```
...
...
[vol_ramp]
name=volume_mute              # When mixer API calls are made,
                              # such as snd_pcm_mixer_write()
duration=20                   # Ramp in the span of 20 milliseconds
profile=20:10,60:80,20:10     # Defined are three linear segments.
                              # Segment 1 specifies 20% of the duration and increase the
                              # volume by 10% of the volume ramp. Segment 2 specifies
                              # 60% of the duration and 80% of the volume increase, and
                              # segment 3 specifies the remaining 20% duration
                              # and to increase the volume by the remaining 10%

[vol_ramp]
name=pause_resume             # Whenever the user calls snd_pcm_*_resume() or
                              # snd_pcm_*_pause()
duration=30                   # duration of 30 milliseconds
                              # No profile key-value is specified, so
                              # it's a linear ramp

[vol_ramp]
name=ducking                  # When ducking policies and preemption is used
duration=60                   # Ramp time is 60 milliseconds
...
...
```

Understanding preemption

Preemption is an optional policy that you can apply to an audio type. It determines whether the audio stream is suspended when its volume is ducked to zero.

The volume can be ducked to zero either because of the ducking policy of a single audio type or the culmulative effect of the ducking policies of multiple higher-priority audio types, same-priority audio types, or both (see "*Understanding the mix policy when using multiple audio types with the same priority*").

You use the audio policy configuration file `preemptable` key to specify whether an audio stream can be preempted.

For example, to stop playing music when a voice call is received, in the audio policy configuration file:

- Configure the voice audio stream with the appropriate priority.
- Set to `true` the `preemption` key for the audio type that applies to the music audio stream.

If an audio stream is ducked to zero and the `preemption` key for the audio type that applies to that stream is not specified or set to `false`, the stream stays in the RUNNING state (SND_PCM_STATUS_RUNNING) while it's muted.

Preemption puts the audio stream into the SUSPENDED (SND_PCM_STATUS_SUSPENDED) state. In some cases, you can use an API call to move out of the SUSPENDED state. For more information, see "*Using APIs to move out of the suspended state*".

Using APIs to move out of the suspended state

When an audio stream is preempted, it moves to the SUSPENDED state (SND_PCM_STATUS_SUSPENDED), which has two modes: soft-suspended and hard-suspended.

The mode determines what happens when you call an API that attempts to transition the stream to the RUNNING mode (for example, *snd_pcm_*_resume()* or *snd_pcm_*_go()*).

soft suspended

Occurs when your audio type is preempted by an audio type of the same priority. If you call an API that transitions the audio stream to the RUNNING state, the suspended audio stream immediately moves from the SUSPENDED to the RUNNING state and preempts any other streams of the same priority.

hard suspended

Occurs when your audio type is preempted by a higher priority audio type. Calling an API that transitions the audio stream to the RUNNING state has no effect while the audio stream is suspended. When the preemption ends, the audio stream moves to the RUNNING state and preempts any other streams of the same priority.

You can use the *ducking_state* member of `snd_pcm_channel_status_t` to determine whether a channel is soft or hard suspended.

When an audio stream is suspended and you call `snd_pcm_*_pause`, the stream remains in the SUSPENDED state until audio concurrency management clears the SUSPENDED state, and then moves to the PAUSED state.

Manually ducking APIs and preemption

If an audio stream is in the PREPARED state (call *snd_pcm_*_prepare()*), you can use the *snd_pcm_channel_audio_ducking()* function to make audio concurrency management apply ducking polices to an audio stream as if it was RUNNING. This call also enables any preemption policy that applies. (See "*Understanding audio ducking*" and *snd_pcm_channel_audio_ducking()*.)

Manual ducking remains active until you either use *snd_pcm_channel_audio_ducking()* to cancel it or reset the parameters using *snd_pcm_plugin_params()*. However, if the stream transitions out of the PREPARE state (for example, to READY), preemption ends but ducking of same- and lower-priority streams continues.

Making the preemption appear transient

You can't explicitly move an audio stream to or out of the SUSPENDED (SND_PCM_STATUS_SUSPENDED) state, but you can define the behavior of an audio stream when the suspension ends.

The `transient` key is set for the interrupting audio stream but configures the behavior of a preempted stream when preemption ends. It makes the interruption appear to be either short-lived (transient) or permanent. When a higher- or same-priority stream with `transient` enabled begins, the lower priority stream is suspended. When the interrupting stream stops, the lower-priority stream automatically resumes playing, which makes the interruption appear short-lived. An interrupting stream with `transient` disabled also suspends a lower priority steam from playing, but when the interrupting stream stops, the lower-priority stream moves to the PAUSED state. The interruption is permanent until *snd_pcm_*_resume()* is called.

When `preemptable` is `true` for its audio type, one of the following transitions occurs for an audio stream when preemption is lifted (no longer ducked to zero):

- If `transient` is `false`, the stream moves from the SUSPENDED state to the PAUSED state (SND_PCM_STATUS_PAUSED) when ducking ends. It requires *snd_pcm_*_resume()* to move back to the RUNNING (SND_PCM_STATUS_RUNNING) state.

  This move to the PAUSED state only happens if a single higher-priority stream ducks the stream to 0. If cumulative ducking sets the volume to 0, the behavior is the same as when `transient` is `true` (see "*Understanding how multiple ducking levels cumulate*").

- If `transient` is `true` the stream moves from the SUSPENDED (SND_PCM_STATUS_SUSPENDED) state back the state it was in before it was preempted.

When `preemptable` is `true:suspend`, the audio stream being preempted transitions back its original state when the suspension ends, regardless of the value of `transient`.

For more information, see "*Syntax of the audio policy configuration file*."

Understanding audio type volume controls

Each audio concurrency management context defined in your system has an associated mixer device created under an audio management card. The mixer device for your audio concurrency management context contains mixer groups for each audio type defined in the context's audio policy configuration file. You can use the standard *snd_mixer_group_*()* APIs to read and write volume and mute settings of these mixer groups, which allows for synchronous adjustments of the volume and mute levels of all audio streams of the specified audio type.

If you have two multimedia subchannels running, adjusting the volume level of the multimedia mixer group synchronously affects the volume levels of both multimedia subchannels.

Additionally these audio management mixer groups have the ability to have their settings saved and restored across power cycles or restart of the `io-audio` service. The volume and mute settings the user had specified can be saved and restored provided you have the following configured on your system:

- `config_write_delay` is set to a non-negative number
- the path **/etc/system/config/audio** is writable on your system
- **deva-util-restore.so** is in the library path on your system

Audio type volume controls are in series with their respective subchannels volume controls, so the levels of both volume controls are factored together to determine the overall user volume level applied to a given audio stream. For example if the volume level for an audio type is set to 50% and the individual subchannel's volume level is set to 25%, the overall volume level of the audio stream would be 12%.

It's important to note that the ducking level applied to a subchannel is scaled by the overall volume level of the ducking subchannel. For example, if the user used the audio type volume controls to reduce the overall volume level of the ducking subchannel by 50% and the configured ducking level was 50%, the final ducking level applied to the ducked subchannels would be 25%.

To illustrate how the various volume levels (subchannel, audio type, and audio ducking) work together, you can use the `mix_ctl` utility as shown below and `cat` the state of the card. In this example, there are two audio streams of audio type `default` and `multimedia` running with the user volume controls (**Subchn** and **AudioType**) configured to full volume (100%). The `multimedia` audio type is ducking the `default` audio type to 50%, so if you use the `cat` command to see the output, you see that the subchannel of `default` audio type has its **Ducking** and **Current** volume level at 50%, while the `multimedia` subchannel is at 100%.

When you use the `cat` command, the channel's information is shown for a card. Here's what some of the parameters mean:

- **Subchn** — The user volume control setting for the specific subchannel.
- **AudioType** — The volume control setting for the specific audio type
- **Control** — The amount of volume control calculated by multiplying the value from the **Subchn** and the **AudioType**.
- **Ducking** — The ducking volume applied to the channel

- **Current** — The calculated resulting volume level based on multiplying the **Control** by **Ducking**

```
# cat /dev/snd/stateC1
McASP @ 0x48468000, McASP
pcmC1D0p
        ocb = 0x10141c38 - subchn = 0x1014ed30
        pid = 1773597  wave @ localhost
                RUNNING - not block - not block - not block
                scount=7483392 used=258048 free=0 frag_size=64512 frags_total=4
                Rate=48000 Format=s16l Voices=2
                NOT MMAPED   CACHED   INTERLEAVED
                pcm_link mode: ASYNC
                Audio Type=default
                Ducking State=ACTIVE
                Volume Ramp Parameters:
                                Subchn  AudioType  Control  Ducking  Current
                   Channel 0     100%       100%     100%     50%      50%
                   Channel 0     100%       100%     100%     50%      50%
        ocb = 0x10150528 - subchn = 0x1014eb78
        pid = 380962    wave @ localhost
                RUNNING - not block - not block - not block
                scount=7483392 used=258048 free=0 frag_size=64512 frags_total=4
                Rate=48000 Format=s16l Voices=2
                NOT MMAPED   CACHED   INTERLEAVED
                pcm_link mode: ASYNC
                Audio Type=multimedia
                Ducking State=ACTIVE
                Volume Ramp Parameters:
                                Subchn  AudioType  Control  Ducking  Current
                   Channel 0     100%       100%     100%     100%     100%
                   Channel 0     100%       100%     100%     100%     100%
```

Now, if you configure the `multimedia` volume level to 50%, you'll see the **AudioType**, **Control** and **Current** values of the `multimedia` subchannel go to 50% and the **Ducking** and **Current** values of the `default` subchannel scaled down to 25%.

```
# mix_ctl -a0:0 group "multimedia" volume=50%
#  cat /dev/snd/stateC1
McASP @ 0x48468000, McASP
pcmC1D0p
        ocb = 0x10141c38 - subchn = 0x1014ed30
        pid = 1773597  wave @ localhost
                RUNNING - not block - not block - not block
                scount=7483392 used=258048 free=0 frag_size=64512 frags_total=4
                Rate=48000 Format=s16l Voices=2
                NOT MMAPED   CACHED   INTERLEAVED
                pcm_link mode: ASYNC
                Audio Type=default
                Ducking State=ACTIVE
                Volume Ramp Parameters:
                                Subchn  AudioType  Control  Ducking  Current
                   Channel 0     100%       100%     100%     25%      25%
                   Channel 0     100%       100%     100%     25%      25%
        ocb = 0x10150528 - subchn = 0x1014eb78
        pid = 380962    wave @ localhost
                RUNNING - not block - not block - not block
                scount=7483392 used=258048 free=0 frag_size=64512 frags_total=4
                Rate=48000 Format=s16l Voices=2
                NOT MMAPED   CACHED   INTERLEAVED
                pcm_link mode: ASYNC
                Audio Type=multimedia
                Ducking State=ACTIVE
                Volume Ramp Parameters:
                                Subchn  AudioType  Control  Ducking  Current
                   Channel 0     100%       50%      50%     100%     50%
                   Channel 0     100%       50%      50%     100%     50%
```

Lastly, if you configure the subchannel volume level of the `multimedia` subchn to 50%, you'll see its **Subchn** value go to 50%, and the **Control** and **Current** values of the `multimedia` subchannel go to 25%. The **Ducking** and **Current** values of the `default` subchannel are further scaled down to 12%, maintaining the overall system audio policy of `default` audio types not exceeding 50% of the `multimedia` volume level.

```
# mix_ctl -a1:0 group "Wave Playback Channel,2" volume=50%
# cat /dev/snd/stateC1
McASP @ 0x48468000, McASP
pcmC1D0p
        ocb = 0x10141c38 - subchn = 0x1014ed30
        pid = 1773597  wave @ localhost
                RUNNING - not block - not block - not block
                scount=7483392 used=258048 free=0 frag_size=64512 frags_total=4
                Rate=48000 Format=s16l Voices=2
                NOT MMAPED   CACHED   INTERLEAVED
                pcm_link mode: ASYNC
                Audio Type=default
                Ducking State=ACTIVE
                Volume Ramp Parameters:
                                Subchn  AudioType  Control  Ducking  Current
                   Channel 0     100%       100%     100%     12%      12%
                   Channel 0     100%       100%     100%     12%      12%
        ocb = 0x10150528 - subchn = 0x1014eb78
        pid = 380962    wave @ localhost
                RUNNING - not block - not block - not block
                scount=7483392 used=258048 free=0 frag_size=64512 frags_total=4
                Rate=48000 Format=s16l Voices=2
                NOT MMAPED   CACHED   INTERLEAVED
                pcm_link mode: ASYNC
                Audio Type=multimedia
                Ducking State=ACTIVE
                Volume Ramp Parameters:
                                Subchn  AudioType  Control  Ducking  Current
```

```
Channel 0     50%       50%       25%      100%     25%
Channel 0     50%       50%       25%      100%     25%
```

Syntax of the audio policy configuration file

Audio policy configuration files have a specific syntax.

- The name is case-insensitive.
- White space within a key or value causes the key-value pair to be discarded, resulting in an invalid configuration file.
- Each key must have a value; a value must have a key.
- The value of the *prio* for the first audio type defined in the configuration doesn't matter since its key has meaning relative to the audio type before it. When you want two audio types to have equal ducking priority, specify `same` for *prio*.
- For each audio type that you define, you must specify all the key-value pairs. If you don't, the configuration isn't valid.
- In general, the first audio type defined in the file is assigned the highest priority.
- If the configuration file isn't valid (e.g., it's missing a required key-value pair for an audio type or an invalid value is specified), `io-audio` fails to start with the resulting information about the failure in the system log.

The following sections are valid:

- `audio_type`: Audio types
- `vol_ramp`: Volume ramping

`[audio_type]` section

The order that you define your audio types using the `[audio_type]` section defines when one audio stream ducks another. For each `audio_type` section, you must use the following key-value pairs to describe the audio type. Every required key must be defined in the audio type section with valid values.

CAUTION: If you leave any of the required key-value pairs out of an `[audio_type]` section, the file is invalid and `io-audio` fails to start.

Below are the required key-value pairs for each audio type you define:

Key: (Required) `name`

A string that specifies the name of the audio type. For example, `phone` or `ringtone`.

Key: (Required) `prio`

The priority directive of the audio type defined as a `String` value. These are the valid values:

- `same` — the audio type has the same priority as previous audio type. If all the audio_types have the `prio` key set to `same`, then all audio types are at the same priority (including the first entry). An entry needs to define `prio=decr` for the priority to change.
- `decr` — the priority of the audio type is one lower than the previously defined audio type.

The first entry you define has the highest priority and forces any lower priority audio streams to duck (lower their volume).

Key: (Required) `duck_same_prio_policy`

The ducking policy applied to audio types of the same priority, defined as a `String` value. **You cannot have different policies configured for audio types with the same priority.** These are the valid values:

- `last_wins` — The last added stream ducks previously added audio streams with the same priority.
- `first_wins` — The first added stream ducks other audio streams with the same priority.
- `mix` — The added audio stream ducks all or specific channels of other streams with the same priority. When you use this policy for audio types with the same priority, those streams are ducked according to the `duck_same_prio_percent` setting. If there are multiple ducking settings are applied, then the strictest (lowest ducking volume) configuration applies. For more information on `duck_same_prio_policy` and `duck_same_prio_percent`, see "*Understanding the `mix` policy when using multiple audio types with the same priority*."

  All audio types with the same priority level and that have `duck_same_prio_policy=mix` must have the same values set for the `duck_lower_prio_percent`.

CAUTION: Don't set `duck_same_prio_policy` to `mix` if you want to use enable `preemption_by_same`. If you do, `io-audio` fails to start.

Key: (Required) `duck_same_prio_percent`

`Integer` or `String:Integer`

The ducking percentage to apply to audio types with the same priority. The value can be specified as either an `Integer`, which specifies a single percentage value, or multiple, comma-delimited `String:Integer` key-value pairs, which specify values for multiple channels from `0` to `7`.

A single value (between 0 or 100) specifies the percentage to reduce the volume by for active streams at the same priority level, for all channels. For example, if you specify `duck_same_prio_percent=50`, it means adjust to 50% of current volume level. To configure no ducking, either specify 100 or the string `noducking`. To understand what happens when multiple percentages are applied, see the "*Understanding the `mix` policy with multiple audio types with the same priority*" section.

To specify percentage values for multiple channels, use the following syntax to specify the channel and the percentage to duck, which delimits each channel with a comma (`,`):

`ch`*X*`:`*ducking_percentage*`,` `ch`*X*`:`*ducking_percentage* where *X* represents the channel number.

It's important to mention that if you don't specify a percentage for an active channel, ducking defaults to 100 (or no ducking).

For example, the following entry specifies that channel 0 adjusts to 50% of its volume level and channel 1 adjusts to 80% of its volume level. Because no value is specified for channel 2, ducking isn't applied to it:

```
duck_same_prio_percent=ch0:50,ch1:80
```

Key: (Required) `duck_lower_prio_percent`

`Integer` or `String:Integer`

The ducking percentage to apply to audio types with a lower priority. Specify one of the following values:

- An `Integer` between 0 and 100, which specifies the percentage to reduce the volume by for active streams of the same priority level, for all channels. For example, a value of 50 (`duck_lower_prio_percent=50`) means adjust to 50% of the current volume level. To configure no ducking, either specify 100 or the string `noducking`.
- Multiple, comma delimited `String:Integer` key-value pairs, which specify a percentage value for one or more specific channels from `0` to `7`.

  Use the following syntax, which delimits each channel with a comma (`,`):

  `ch`*X*`:`*ducking_percentage*, `ch`*X*`:`*ducking_percentage*

  where *X* represents the channel number.

  It's important to mention that if you don't specify a percentage for an active channel, ducking defaults to 100 (or no ducking).

  For example, the following specifies that channel 0 adjusts to 50% of its volume level and channel 1 adjusts to 80% of its volume level. Because no value is specified for channel 2, ducking isn't applied to it (effectively 100):

  ```
  duck_same_prio_percent=ch0:50,ch1:80
  ```

Key: (Optional) `preemptable`

`String` (`true`, `true:suspend`, or `false`)

(Optional) This key specifies whether the audio type can be preempted by audio concurrency management. The default behavior when this key isn't specified is `false`. You can specify one of the following values for this key:

- `false` — Indicates to audio concurrency management that this audio stream can't be preempted. However, you can still explicitly call APIs to transition to the PAUSED state.

  It's important to remember that when the audio stream is ducked to zero (that is, muted), the audio stream remains in the RUNNING (SND_PCM_STATUS_RUNNING) state.

- `true` — Indicates to audio concurrency management that the audio type can preempted. The behavior when the preemption ends depends on the value of the `transient` key for the audio type that imposes a ducking policy. If `transient` is set to `true` for the preempting audio stream, then the preempted audio stream moves to the SUSPENDED state when preemption occurs and transitions back to its previous state when the preemption ends (no longer ducked to zero). If `transient` is `false` for the preempting audio stream, then the preempted audio stream transitions to the SUSPENDED state but transitions to the PAUSED state when the preemption ends. Because your audio stream is in the PAUSED state, you must call *snd_pcm_*_resume()* to resume playing.

- `true:suspend` — Indicates to audio concurrency management that the audio type can be preempted, but is limited to transitioning to the SUSPENDED state and then back to its original state. It's important to mention that audio concurrency management won't transition the audio stream to the PAUSED state. This setting is useful for audio types where you don't want to call *snd_pcm_*_resume()* to resume playing the audio stream.

Key: (Optional) `transient`

`Boolean` (`true` or `false`)

(Optional) This key specifies the state that audio concurrency management moves a preempted audio type to when a preemption ends:

- `true` — This audio type can cause a preemptable audio type to move to the SUSPENDED state and when the preemption ends, the preempted audio stream transitions back to its original state.
- `false` — This audio type can cause a preemptable audio type to move to the SUSPENDED state and when preemption ends, it moves to the PAUSED state. This means that you must call *snd_pcm_*_resume()* to resume playing the audio stream.

  It's important to mention that when this key is set to `false`, but the `preemptable` key is set to `true:suspend`, the audio stream that was preempted moves back to its original state when preemption is lifted (no longer ducked to zero).

Audio types of the same priority level and with the `duck_same_prio_policy` set to `mix` must have the same setting for this key, otherwise an error occurs when loading the audio policy configuration file.

(Optional) `[vol_ramp]` section

This section permits you to define volume ramping for audio concurrency management scenarios based on specific names. The `name` and `duration` are required key-value pairs, and `profile` is optional. If you don't define ramping for an audio concurrency management scenario, the default duration is 20 milliseconds with a linear volume ramp (i.e., 100:100).

For each volume ramp (`[vol_ramp]`) section, you can use the following key-value pairs to describe the audio type.

Key: (Required) `name`

`String`(`ducking`, `pause_resume`, `volume_mute`)

This is a profile name that represents the scenario when ramping occurs. You can use one of these predefined names:

- `ducking` — This name corresponds to any changes that are made due to ducking policies being used.

- `pause_resume` — This name corresponds when the audio stream is paused or resumed using API calls.
- `volume_mute` — This corresponds to when volume changes are made, including muting using the mixer. You can override this behavior using API calls (e.g., *snd_mixer_group_write()* and the `change_duration` member in the *`snd_mixer_group_t`*). If API calls don't specify the duration, it's treated as a zero, and the duration configured for this scenario is used for volume ramping.

Key: (Required) `duration`

`Integer`

This number represents the number of milliseconds to ramp to the specified volume, called ramp duration. This is the time it takes for the sum of all ramp segments to complete. For `ducking` and `pause_resume` scenarios, this parameter indicates the duration to complete ramping. The value specified here is used if the duration isn't specified

Key: (Optional) `profile`

`Integer:Integer`(sample percentage, volume increase percentage)

When specified, the number of sample pairs (comma-delimited pairs) represents the number of ramp segments. Each ramp segment indicates a percentage of the sample (duration) and the percentage of the volume delta. The sum of the percentage values for the ramp samples and volume increment values must equal to `100`.

For example:

```
profile=20:10,60:80,20:10
```

In the example above, there are three ramp segments, which create a non-linear volume ramp. The segments mean:

- For the first 20% of the sample (duration), increase the volume by 10% of the volume delta.
- For the next 60% of the sample, increase the volume by 80% of the volume delta.
- For the remaining 20% of the sample, increase to the remaining volume delta.

Notice that the sum of the percentage samples(duration) values (20 + 60 + 20) is `100`; likewise, the sum of percentage values to increment the volume by (10 + 80 + 10) is also `100`.

Example of an audio policy configuration file

```
[audio_type] # section defining an audio type
name=voice                                 # Audio type name string
prio=same                                  # Priority directive of the audio type,
                                           # possible value is "same" or "decr"
duck_same_prio_policy=last_wins            # Ducking policy of the same priority type,
                                           # Possible values are "last_wins",
                                           # "first_wins", or "mix"
duck_same_prio_percent=0                   # Ducking percentage applied to same
                                           # priority audio types
duck_lower_prio_percent=0                  # Ducking percentage applied to lower
                                           # priority audio types
```

```
transient=false                            # Cause lower preemptable types to suspend

[audio_type]
name=ringtone
prio=decr                                  # Priority of the audio type is decreased
                                           # by one from previous audio type
duck_same_prio_policy=last_wins            # Last active audio stream that can duck
                                           # previous audio streams of same priority
duck_same_prio_percent=50                  # all channels of active streams at the
                                           # same priority level are ducked to 50 percent
duck_lower_prio_percent=50                 # all channels of active streams at lower
                                           # priority levels are ducked to 50 percent

[audio_type]
name=nav
prio=decr
duck_same_prio_policy=last_wins
duck_same_prio_percent=ch0:50,ch1:100      # Channel/Voice 0 ducked to 50 percent,
                                           # Channel/Voice 1 remains at 100 percent
duck_lower_prio_percent=ch0:50,ch1:100     # Ducking values will default to 100
                                           # (no ducking) for channels not listed

[audio_type]
name=tts_front
prio=decr
duck_same_prio_policy=last_wins
duck_same_prio_percent=80
duck_lower_prio_percent=80

[audio_type]
name=tts_back
prio=same                                  # Priority of the audio type is the
                                           # same as the previously defined type
duck_same_prio_policy=last_wins
duck_same_prio_percent=0
duck_lower_prio_percent=50

# To allow games and media to play concurrently, define the game type one above
# media with duck_lower_prio_percent=100/noducking
# Note: This means that if there are other lower priority types
#defined they will also not be ducked by the game type.

[audio_type]
name=game
prio=decr
duck_same_prio_policy=mix                  # Mix audio streams at same priority with
                                           # the below ducking configuration
duck_same_prio_percent=100                 # 100 percent
duck_lower_prio_percent=noducking
preemptable=true:suspend                   # Suspend only if preempted or ducked to 0
# Live media is not pausable,
# so preemption is not enabled
[audio_type]
name=live_multimedia
```

```
prio=decr
duck_same_prio_policy=last_wins
duck_same_prio_percent=0
duck_lower_prio_percent=70


# The only difference between the live_multimedia and multimedia types
# is that multimedia can be paused so that preemption is enabled
[audio_type]
name=multimedia
prio=same
duck_same_prio_policy=last_wins
duck_same_prio_percent=0
duck_lower_prio_percent=70
preemptable=true                             #pause or suspend when preempted

[audio_type]
name=default
prio=decr
duck_same_prio_policy=mix                   # Mix audio streams at same priority
                                            # with the below ducking configuration
duck_same_prio_percent=100                  # 100 percent of volume (no ducking)
duck_lower_prio_percent=noducking
preemptable=true:suspend                    # Audio stream is suspended when
                                            # ducked to 0 percent by higher priority
                                            # types
[vol_ramp]
name=volume_mute                            # When mixer API calls are made,
                                            # such as snd_pcm_mixer_write(),
duration=20                                 # ramp in the span of 20 milliseconds
profile=20:10,60:80,20:10                   # Three linear segments are defined:
                                            # Segment 1 specifies 20% of the duration
                                            # and increases the volume by 10% of the
                                            # volume ramp. Segment 2 specifies 60% of
                                            # the duration and 80% of the volume
                                            # increase. Segment 3 specifies the
                                            # remaining 20% duration and to increase
                                            # the volume by the remaining 10%.

[vol_ramp]
name=pause_resume                           # Whenever the user calls snd_pcm_*_resume() or
                                            # snd_pcm_*_pause()
duration=30                                 # duration of 30 milliseconds
                                            # No profile key-value is specified, so
                                            # it's a linear ramp

[vol_ramp]
name=ducking                                # When ducking policies and preemption is used
duration=60                                 # Ramp time is 60 milliseconds
```

Chapter 5
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()*:

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

- In the client application, configure the audio interface to use a smaller audio fragment size. If playback is using the PCM software mixer, then the fragment size can be any multiple of the PCM software mixer fragment size. By default this is 1024 samples per channel. So if your audio configuration uses 16-bit stereo (2 channel) then the fragment size would be 4KB. Here's how the fragment size is calculated:

  ```
  fragment size (1024 samples per channel in this case) x 2 bytes per sample (16-bit)
                         x 2 channels
                         = 4096 bytes
  ```

  Make sure to look at the fragment size returned via the *snd_pcm_plugin_setup()* 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, sample size (configured using the `sw_mixer_samples` option or `sw_mixer_ms`), potentially required data conversions, and so on. For more information, see "*PCM software mixer*"

- 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()*) 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.
- The PCM Software Mixer is automatically used unless you explicitly disable it. When the PCM Software is enabled (default behavior), 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. Otherwise, if the PCM Software Mixer is disabled, you can wait until two audio fragments are written before issuing the `go` command.

Chapter 6
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. Although the function names may be the same, there's no guarantee that QSA and ALSA calls have the same behavior, and some definitely don't.

All the documented APIs are available for you to use with the exception of the *snd_afm_*()* and *snd_apx_*()* functions, which are enabled only if you have installed QNX Acoustic Management Platform 3.0.

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_afm_close()

Close an AFM handle and free its resources

Synopsis:

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

int snd_afm_close( snd_afm_t *handle );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_close()* function frees all resources allocated with the AMP Functional Module (AFM) handle and closes the connection to the AFM interface.

Returns:

EOK on success, or a negative *errno* upon failure.

This function can also return the return values of *close()* (see *close()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_open() (p. 110)

Create a handle and open a connection to a specified AMP Functional Module (AFM)

snd_afm_open_name() (p. 112)

Create a handle and open a connection to an AMP functional module specified by name

snd_afm_file_descriptor()

Return the file descriptor of the connection to the AFM interface

Synopsis:

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

int snd_afm_file_descriptor( snd_afm_t *handle );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_file_descriptor()* function returns the file descriptor of the connection to the Acoustics Management Platform Functional Module (AFM) interface.

You can use this file descriptor with *ioctl()*, *ionotify()*, *poll()*, or *select()* (see the QNX Neutrino *C Library Reference*).

Don't *close()* this file descriptor.

Returns:

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

This function can also return the return values of *close()* (see *close()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The *handle* is invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	Yes
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

in the QNX Neutrino *C Library Reference*

snd_afm_get_ap_data()

Retrieve data from the acoustic library in an AFM

Synopsis:

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

int snd_afm_get_ap_data( snd_afm_t *handle,
                         snd_afm_ap_param_t *param,
                         void *data );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

param

A pointer to a `snd_afm_ap_param_t` structure that specifies the library parameter information and the size of the data buffer.

data

NULL, or a pointer to the location where the function can store the retrieved data.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_get_ap_data()* function retrieves data from the acoustic library in an AFM, as specified in the `snd_afm_ap_param_t` structure that *param* points to:

```
typedef struct snd_afm_ap_param {
    uint32_t size;
    uint32_t dataId;
    int32_t  channel;
    int32_t  status;
} snd_afm_ap_param_t;
```

This structure includes the following members:

size

The size of the buffer that *data* points to. On return, this member is updated with the actual size of the retrieved data.

dataId

The identifier for the parameter whose value you want to get. Each AFM has its own set of IDs; see the AFM's **<qwa_*afm*_defs.h>** header file (e.g., **<qwa_icc_defs.h>**). The comments for each parameter indicate the data type and when it's valid to get and set the parameter.

channel

The channel identifier, which is applicable only for selected channel-based parameters (see the AFM's documentation). The channel number is zero-based and should be from 0 to one less than the maximum number of channels valid for the specified parameter.

status

The return code from the library; one of the QWA_* codes defined in **<qwa_err.h>**.

If you specify NULL for *data*, no data is returned, but the *size* member is set to the minimum size required for the data buffer.

This function is applicable only to acoustic (ASD, CSA, ICC, QAV) AFMs. For more information about getting and setting parameters, see the documentation for the specific AFM.

Returns:

EOK if the request was successfully issued to the library and a response was returned, or a negative *errno* value if an error occurred.

You should check both the function's return value and the *status* member of the `snd_afm_ap_param_t` structure to determine if a call is successful.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The value of *handle* or *param* is NULL.

Classification:

QNX Neutrino

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_load_ap_dataset() (p. 108)
Load an acoustic processing data set into a running AFM

snd_afm_set_ap_data() (p. 114)
Set data within the acoustic library in an AFM

snd_afm_get_audio_mode()

Get the currently applied audio mode

Synopsis:

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

int snd_afm_get_audio_mode( snd_afm_t *handle,
                            char *mode,
                            int size );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

mode

A buffer where the function can store the mode.

size

The size of the *mode* buffer, in bytes.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_get_audio_mode()* function gets the currently applied audio mode that's used to tune acoustic processing parameters.

Returns:

EOK on success, or a negative *errno* value upon failure.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The value of *handle* or *mode* is NULL.

-ENOMEM

The *mode* buffer isn't big enough to hold the mode.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_set_audio_mode() (p. 117)
Set the audio mode used to tune acoustic processing parameters

snd_afm_get_param()

Get a configuration parameter for the specified AFM

Synopsis:

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

int snd_afm_get_param( snd_afm_t *handle,
                       int id);
                       size_t* size);
                       void* data);
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

id

The identifier for the AFM parameter to get the value for.

size

The size of the buffer that *data* points to.

data

A pointer to the location to store the retrieved value in.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_get_param()* function is only used with specific AFMs. For information on which parameters are available for a specific AFM and their expected sizes, see the corresponding AFM header installed under **usr/include/ado_afm/**.

Returns:

EOK on success, or a negative *errno* upon failure.

Errors:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_set_param() (p. 119)
Set a configuration parameter for the specified AFM

snd_afm_get_vin_list()

Get the current Vehicle Input (VIN) list from an AFM

Synopsis:

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

int snd_afm_get_vin_list( snd_afm_t *handle,
                          snd_afm_vin_list_item_t *list,
                          int num );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

list

An array of `snd_afm_vin_list_item_t` entries to populate; see below.

num

The number of entries in the array. Use *snd_afm_get_vin_list_count()* to determine the number of entries required.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_get_vin_list_count()* function gets the list of vehicle inputs that you should retrieve data for from the CAN bus or other vehicle interface and inputs it to the AFM using *afm_set_win_stream()*. The number and type of vehicle inputs depends on the AFM's current configuration and can only be retrieved after the AFM is running.

The `afm_vin_list_item` structure is defined as follows:

```
typedef struct snd_afm_vin_list_item
{
   int16_t key;
   int16_t min;
```

```
    int16_t max;
    int16_t is_rpm;
} snd_afm_vin_list_item_t;
```

The members include:

key

A unique key that's used to identify this parameter.

min

The minimum value that the parameter can take.

max

The maximum value that the parameter can take.

is_rpm

1 if the ASD AFM should treat this parameter as the RPM input; 0 otherwise.

Returns:

EOK on success, or a negative *errno* value upon failure.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EAGAIN

The AFM hasn't been started.

-EINVAL

The value of *handle* or *list* is NULL.

-ENOMEM

The array needs more than *num* entries.

-ENOTSUP

The AFM doesn't support VIN lists.

Classification:

QNX Neutrino

Safety:	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_vin_list_count() (p. 102)
: *Get the number of Vehicle Input (VIN) items required by an AFM's VIN list*

snd_afm_set_vin_stream() (p. 121)
: *Set the current Vehicle Input (VIN) data for an AFM.*

snd_afm_get_vin_list_count()

Get the number of Vehicle Input (VIN) items required by an AFM's VIN list

Synopsis:

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

int snd_afm_get_vin_list_count( snd_afm_t *handle,
                                int *num );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

num

A pointer to a location where the function can store the number.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_get_vin_list_count()* function gets the number of Vehicle Input (VIN) items defined for the AFM's current configuration. You can use the returned number when you allocate the buffer to retrieve the VIN list with *snd_afm_get_vin_list()* and set the VIN data with *snd_afm_set_vin_stream()*. The number and type of vehicle inputs depends on the AFM's current configuration and can only be retrieved once the AFM is running.

Returns:

EOK on success, or a negative *errno* value upon failure.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EAGAIN

The AFM hasn't been started.

-EINVAL

The value of *handle* or *num* is NULL.

-ENOTSUP

The AFM doesn't support VIN lists.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_vin_list() (p. 99)
Get the current Vehicle Input (VIN) list from an AFM

ssnd_afm_set_vin_stream() (p. 121)
Set the current Vehicle Input (VIN) data for an AFM.

snd_afm_info()

Get the setup information for an AFM

Synopsis:

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

int snd_afm_info( snd_afm_t *handle,
                  snd_afm_info_t *info);
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

info

A pointer to the location to store the retrieved AFM information in.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_info()* function retrieves the setup information for an AMP Functional Module (AFM) with the specified handle.

Returns:

The setup information for the specified AFM on success, or a negative *errno* upon failure.

Errors:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

`snd_afm_info_t` (p. 106)
: *Setup information about an AFM*

snd_afm_status() (p. 126)
: *Get the current status of an AFM*

snd_afm_info_t

Setup information about an AFM

Synopsis:

```
#define SND_AFM_INFO_ROUTE_MAIN 0x0001
#define SND_AFM_INFO_START_ON_BOOT 0x0002
#define SND_AFM_INFO_PASS_THROUGH 0x0004
#define SND_AFM_INFO_VOL_CTRL 0x0008

typedef struct snd_afm_info
{
    int32_t     card;
    int32_t     device;
    char        name[80];
    char        cardname[80];
    uint32_t    flags;
}snd_afm_info_t;
```

Description:

The `snd_afm_info_t` structure describes an AMP Functional Module (AFM). To get this information, call *snd_afm_info()*.

The members include:

card

The sound card number.

device

The device number on the card.

name

The symbolic name associated with the device.

cardname

The name of the card.

flags

A bitwise union of SND_AFM_INFO_* information types, which correspond to values set in the `[AFM]` section of the audio configuration file:

- SND_AFM_INFO_PASS_THROUGH — The `enable_pass_through` key is `1`.
- SND_AFM_INFO_ROUTE_MAIN — The `sw_mixer_rout` key is `main`.
- SND_AFM_INFO_START_ON_BOOT — The `start_on_boot` key is `1`.

- SND_AFM_INFO_VOL_CTRL — The `enable_vol_ctl` key is `1`.

For more information, see “Audio configuration file” in the entry for `io_audio` in the *Utilities Reference*.

Classification:

QNX Neutrino

Related Links

snd_afm_info() (p. 104)
Get the setup information for an AFM

snd_afm_load_ap_dataset()

Load an acoustic processing data set into a running AFM

Synopsis:

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

int snd_afm_load_ap_dataset( snd_afm_t *handle,
                             const char *dataset,
                             int *ap_status );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

dataset

The name of the data set to load.

ap_status

A pointer to a location where the function can store the acoustic library return code.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_load_ap_dataset()* function specifies the name of an acoustic processing parameter data set that can be used to qualify the parameter tuning specified with *snd_afm_set_audio_mode()*. The *dataset* name is appended to the string `ap_dataset_qcf_` to create a **.conf** key used to look up the acoustic dataset file path. An error is returned if the dataset file can't be found; otherwise, the name of the dataset is stored and one of the following actions is taken:

- If the AFM is running, the data set is loaded immediately. The library return code is returned in the *ap_status* argument. An application should check both the return value of *snd_afm_load_ap_dataset()* and the *ap_status* argument to determine if a call is successful. Passing an empty dataset while running returns the error EBUSY.

- If the AFM isn't running, the data set is loaded when the AFM is started, after the library is initialized with the qcf configuration file for the specified mode, but before processing is started. If an error occurs when loading the data set, the AFM continues to run. Passing an empty dataset while idle clears the stored dataset and no dataset is loaded on the next start.

Once set, the data set is loaded each time the AFM is started until a new data set is applied.

This function is applicable only to acoustic (ASD, CSA, ICC, QVP) AFMs.

Returns:

EOK on success, or a negative *errno* value upon failure.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The value of *handle*, *dataset*, or *ap_status* is NULL.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_ap_data() (p. 92)
Retrieve data from the acoustic library in an AFM

snd_afm_set_ap_data() (p. 114)
Set data within the acoustic library in an AFM

snd_afm_set_audio_mode() (p. 117)
Set the audio mode used to tune acoustic processing parameters

snd_afm_open()

Create a handle and open a connection to a specified AMP Functional Module (AFM)

Synopsis:

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

int snd_afm_open( snd_afm_t **handle,
                  int card,
                  int device );
```

Arguments:

handle

A pointer to a location where *snd_afm_open()* stores a handle for the AFM

card

The card number.

device

The audio device number.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_open()* function creates a handle and opens a connection to the Acoustics Management Platform (AMP) Functional Module (AFM) for sound card number *card* and audio device number *device*. AFMs are installed in **/dev/snd** and their names are in the form `afmC`*x*`D`*y*, where *x* is the card number, and *y* is the device number. The card number depends on the order in which the cards are specified in the `io-audio` **.conf** file.

There are no defaults; your application must specify all the arguments to this function.

Using names for audio devices (*snd_afm_open_name()*) is preferred to using numbers (*snd_afm_open()*).

Returns:

Zero on success, or a negative error code.

This function can also return the negative of the values that *open()* can assign to *errno* (see *open()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The card number or the device number is invalid.

-ENOMEM

There wasn't enough memory to allocate control structures.

-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	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_close() (p. 88)
Close an AFM handle and free its resources

snd_afm_open_name() (p. 112)
Create a handle and open a connection to an AMP functional module specified by name

snd_afm_open_name()

Create a handle and open a connection to an AMP functional module specified by name

Synopsis:

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

int snd_afm_open_name( snd_afm_t **handle,
                       char *filename );
```

Arguments:

handle

A pointer to a location where *snd_afm_open_name()* can store a handle for the AMP functional module. You'll need this handle when you call the other *snd_afm_** functions.

filename

The name of the AFM to open. Can be one of the following names (all found under **/dev/snd**):

- a symbolic name (for example, `voice`, or a name specified by the `sym_name` key in the audio configuration file).
- a system-assigned name in the form `afmC`*x*`D`*y*, where *x* is the card number, and *y* is the device number.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_open_name()* function creates a handle and opens a connection to the named AMP Functional Module (AFM). AFMs are installed in **/dev/snd** and their names are in the form `afmC`*x*`D`*y*, where *x* is the card number, and *y* is the device number. The card number depends on the order in which the cards are specified in the `io-audio` **.conf** file.

Each AFM also has a symbolic name associated with it (e.g., `icc`, `voice`, `asd`, and `csa`). You can use the `sym_name` key in the audio configuration file to add a symbolic name (which is useful if your configuration has multiple AFMs of the same type).

Using names for AFMs (*snd_afm_open_name()*) is preferred to using numbers (*snd_afm_open()*).

Returns:

EOK on success, or a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

This function can also return the negative of the values that *open()* can assign to *errno* (see *open()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The *filename* is NULL or an empty string.

-ENOMEM

There wasn't enough memory to allocate control structures.

-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	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_close() (p. 88)
Close an AFM handle and free its resources

snd_afm_open() (p. 110)
Create a handle and open a connection to a specified AMP Functional Module (AFM)

snd_afm_set_ap_data()

Set data within the acoustic library in an AFM

Synopsis:

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

int snd_afm_set_ap_data( snd_afm_t *handle,
                         snd_afm_ap_param_t *param,
                         const void *data );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

param

A pointer to a `snd_afm_ap_param_t` structure that specifies the library parameter information and the size of the data buffer.

data

A pointer to the new value for the parameter.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_set_ap_data()* function retrieves data from the acoustic library in an AFM, as specified in the `snd_afm_ap_param_t` structure that *param* points to:

```
typedef struct snd_afm_ap_param {
    uint32_t size;
    uint32_t dataId;
    int32_t  channel;
    int32_t  status;
} snd_afm_ap_param_t;
```

This structure includes the following members:

size

The size of the buffer that *data* points to.

dataId

The identifier for the parameter whose value you want to set. Each AFM has its own set of IDs; see the AFM's **<qwa_*afm*_defs.h>** header file (e.g., **<qwa_icc_defs.h>**). The comments for each parameter indicate the data type and when it's valid to get and set the parameter.

channel

The channel identifier, which is applicable only for selected channel-based parameters (see the AFM's documentation). The channel number is zero-based and should be from 0 to one less than the maximum number of channels valid for the specified parameter.

status

The return code from the library; one of the QWA_* codes defined in **<qwa_err.h>**.

This function is applicable only to acoustic (ASD, CSA, ICC, QAV) AFMs. For more information about getting and setting parameters, see the documentation for the specific AFM.

If volume control is enabled (Voice and ICC AFMs), don't use this API to change the volume and muting in the library, since the changes won't be reflected to the mixer control.

Returns:

EOK if the request was successfully issued to the library and a response was returned, or a negative *errno* value if an error occurred.

You should check both the function's return value and the *status* member of the `snd_afm_ap_param_t` structure to determine if a call is successful.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The value of *handle*, *param*, or *data* is NULL.

Classification:

QNX Neutrino

Safety:	
Cancellation point	Yes

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_ap_data() (p. 92)
Retrieve data from the acoustic library in an AFM

snd_afm_load_ap_dataset() (p. 108)
Load an acoustic processing data set into a running AFM

snd_afm_set_audio_mode()

Set the audio mode used to tune acoustic processing parameters

Synopsis:

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

int snd_afm_set_audio_mode( snd_afm_t *handle,
                            const char *mode );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

mode

A string containing the mode.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_set_audio_mode()* function sets the audio mode that's used to tune acoustic processing parameters.

The *mode* string is appended to `ap_qcf_` to create a `.conf` key used to look up the acoustic tuning file path. Passing an empty string sets the mode back to `default`. An error is returned and the mode remains unchanged if the key can't be found in the `.conf` or if the associated path can't be found in the filesystem.

This function changes the path to the qcf file that's loaded when the AFM is started, or when either PCM capture or playback is started on **/dev/snd/voice**.

You should call *snd_afm_set_audio_mode()* only when the AFM is stopped or idle.

Returns:

EOK on success, or a negative *errno* value upon failure.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The value of *handle* or *mode* is NULL.

-ENOENT

The `ap_qcf_*` key doesn't exist in the configuration file.

-ENOTSUP

The AFM doesn't support setting the audio mode.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_audio_mode() (p. 95)
Get the currently applied audio mode

snd_afm_ap_load_dataset() (p. 108)
Load an acoustic processing data set into a running AFM

snd_afm_set_param()

Set a configuration parameter for the specified AFM

Synopsis:

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

int snd_afm_set_param( snd_afm_t *handle,
                       int id);
                       size_t size);
                       const void* data);
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

id

The identifier for the AFM parameter to set the value of.

size

The size of the buffer that *data* points to.

data

A pointer to the location of the value to set.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_set_param()* function is used only with specific AFMs. For information on which parameters are available for a specific AFM and their expected sizes, see the corresponding AFM header installed under **usr/include/ado_afm/**.

Returns:

EOK on success, or a negative *errno* upon failure.

Errors:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_param() (p. 97)

Get a configuration parameter for the specified AFM

snd_afm_set_vin_stream()

Set the current Vehicle Input (VIN) data for an AFM.

Synopsis:

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

int snd_afm_set_vin_stream( snd_afm_t *handle,
                            snd_afm_vin_pair_t *stream,
                            int num );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

stream

An array of `snd_afm_vin_pair_t` entries to set; see below.

num

The number of entries in the array.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_set_vin_stream* function sets the current Vehicle Input data for an AFM. This function should be called when the VIN data changes while the AFM is running.

The stream should be an array of `snd_afm_vin_pair_t` structures corresponding to the VINS retrieved with *snd_afm_get_vin_list()*. The `snd_afm_vin_pair_t` structure is defined as follows:

```
typedef struct snd_afm_vin_pair
{
   int16_t key;
   int16_t value;
} snd_afm_vin_pair_t;
```

The members include:

key

A unique key that's used to identify this parameter.

value

The current value of this parameter to send to the AFM(s).

The *snd_afm_set_vin_stream()* function is used to send vehicle information (e.g., RPM, vehicle speed, etc.) to the ASD AFMs. This information is expected to change rapidly, and this API provides the means to pass data to the AFMs regularly while they're processing. The stream should contain one `snd_afm_vin_pair_t` for each `afm_vin_list_item_t` returned from *snd_afm_get_vin_list()*, with the keys set to match. The value should be between the minimum and maximum values returned in the `afm_vin_list_item_t`.

Returns:

EOK on success, or a negative *errno* value upon failure.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-E2BIG

The *stream* array is too big.

-EAGAIN

The AFM isn't running.

-EINVAL

The value of *handle* or *stream* is NULL.

-ENOTSUP

The AFM doesn't support VIN lists.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_vin_list() (p. 99)

Get the current Vehicle Input (VIN) list from an AFM

snd_afm_get_vin_list_count() (p. 102)

Get the number of Vehicle Input (VIN) items required by an AFM's VIN list

snd_afm_start()

Start AFM processing

Synopsis:

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

int snd_afm_start( snd_afm_t *handle );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_start()* function starts AFM audio or ACS link processing.

Returns:

EOK on success, or a negative *errno* value upon failure.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EBUSY

One of the following:

- The AFM or ACS link has already been started.

-EINVAL

The value of *handle* is NULL.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_audio_mode() (p. 95)
Get the currently applied audio mode

snd_afm_get_vin_list() (p. 99)
Get the current Vehicle Input (VIN) list from an AFM

snd_afm_get_vin_list_count() (p. 102)
Get the number of Vehicle Input (VIN) items required by an AFM's VIN list

snd_afm_set_vin_stream() (p. 121)
Set the current Vehicle Input (VIN) data for an AFM.

snd_afm_set_audio_mode() (p. 117)
Set the audio mode used to tune acoustic processing parameters

snd_afm_load_ap_dataset() (p. 108)
Load an acoustic processing data set into a running AFM

snd_afm_stop() (p. 129)
Stop AFM processing

snd_afm_status()

Get the current status of an AFM

Synopsis:

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

int snd_afm_status( snd_afm_t *handle,
                    snd_afm_status_t *status);
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

status

A pointer to the location to store the retrieved AFM status in.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_status()* function retrieves the status information for an AMP Functional Module (AFM) with the specified handle.

Returns:

The status for the specified AFM on success, or a negative *errno* upon failure.

Errors:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_info() (p. 104)
Get the setup information for an AFM

`snd_afm_status_t` (p. 128)
The status of an AFM

snd_afm_status_t

The status of an AFM

Synopsis:

```
#define SND_AFM_STATE_RUNNING 0x01
#define SND_AFM_STATE_RUNNING_PCM 0x02
#define SND_AFM_STATE_IDLE 0x03
#define SND_AFM_STATE_SHUTDOWN 0x04

typedef struct snd_afm_status
{
    uint64_t    ms_processed;
    uint32_t    state;
}snd_afm_status_t;
```

Description:

The `snd_afm_status_t` structure describes the status of an AMP Functional Module (AFM). To get this information, call *snd_afm_status()*.

The members include:

ms_processed

The number of milliseconds of audio that the AFM has processed since it was started.

state

One of the SND_AFM_STATE_* state types:

- SND_AFM_STATE_RUNNING — AFM is running using audio provided by a hardware device.
- SND_AFM_STATE_RUNNING_PCM — AFM is running using audio provided through its software PCM interface.
- SND_AFM_STATE_SHUTDOWN — AFM is shut down.
- SND_AFM_STATE_IDLE — AFM is idle.

Classification:

QNX Neutrino

Related Links

snd_afm_status() (p. 126)
Get the current status of an AFM

snd_afm_stop()

Stop AFM processing

Synopsis:

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

int snd_afm_stop( snd_afm_t *handle );
```

Arguments:

handle

The handle for the AFM device, which you must have opened by calling *snd_afm_open_name()* or *snd_afm_open()*.

Library:

libasound

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_afm_stop()* function stops AFM audio or ACS link processing. No error is returned if the AFM is already stopped.

Returns:

EOK on success, or a negative *errno* value upon failure.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

-EINVAL

The value of *handle* is NULL.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_afm_t`) is used across multiple threads.

Related Links

snd_afm_get_audio_mode() (p. 95)
Get the currently applied audio mode

snd_afm_get_vin_list() (p. 99)
Get the current Vehicle Input (VIN) list from an AFM

snd_afm_get_vin_list_count() (p. 102)
Get the number of Vehicle Input (VIN) items required by an AFM's VIN list

snd_afm_set_vin_stream() (p. 121)
Set the current Vehicle Input (VIN) data for an AFM.

snd_afm_set_audio_mode() (p. 117)
Set the audio mode used to tune acoustic processing parameters

snd_afm_load_ap_dataset() (p. 108)
Load an acoustic processing data set into a running AFM

snd_afm_start() (p. 124)
Start AFM processing

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 with `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

Related Links

snd_card_name() (p. 135)
Find the card number for a given name

snd_card_get_name() (p. 133)
Find the name for a given card number

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 with `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

Related Links

snd_card_get_longname() (p. 131)
Find the long name for a given card number

snd_card_name() (p. 135)
Find the card number for a given name

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 with `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

Related Links

snd_card_get_longname() (p. 131)
Find the long name for a given card number

snd_card_get_name() (p. 133)
Find the name for a given card number

snd_cards()

Count the sound cards

Synopsis:

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

int snd_cards ( void );
```

Library:

libasound.so

Use the `-l asound` option with `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()* instead.

Returns:

The number of sound cards.

Classification:

QNX Neutrino

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

Related Links

snd_cards_list() (p. 138)
Count the sound cards and list their card numbers in an array

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 with `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()* 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

Related Links

snd_cards() (p. 137)
Count the sound cards

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 (*audiomgmt) (snd_ctl_t *handle, void *private_data, int cmd);
    void (*afm) (void *private_data, int cmd, snd_ctrl_afm_event_t* event);
    void *reserved[27]; /* 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()*.

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. The following is the only argument for the callback:

private_data

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

xswitch callback

The *xswitch* callback is called whenever a switch changes. The following are the arguments for this callback:

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:

- 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.

audiomgmt **callback**

The *audiomgmt* callback is called whenever the list of audio types changes. Its arguments include the following:

handle

A handle to control device.

private_data

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

cmd

One of:

- SND_CTL_READ_AUDIOMGMT_CHG (Deprecated)
- SND_CTL_READ_AUDIOMGMT_STATUS_CHG

afm **callback**

The *afm* callback is called whenever there is an AFM event. Its arguments include the following:

private_data

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

cmd

The type of event. One of:

- SND_CTL_READ_AFM_AUDIO_MODE
- SND_CTL_READ_AFM_DATASET
- SND_CTL_READ_AFM_DETECT
- SND_CTL_READ_AFM_STATE

event

A pointer to a `snd_ctrl_afm_event_t` structure that provides details about the event.

Classification:

QNX Neutrino

Related Links

snd_ctl_read() (p. 170)
Read pending control events

snd_ctl_ducking_status_read() (p. 149)
Get all active audio subchannels associated with an audio ducking output

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()*.

Library:

libasound.so

Use the `-l asound` option with `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

Related Links

snd_ctl_open() (p. 164)

Create a connection and handle to the specified control device

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()*.

Library:

libasound.so

Use the `-l asound` option with `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()* 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

Safety:	
Signal handler	Yes
Thread	Yes

Related Links

snd_ctl_open() (p. 164)

Create a connection and handle to the specified control device

snd_ctl_read() (p. 170)

Read pending control events

in the QNX Neutrino *C Library Reference*

snd_ctl_ducking_read()

Get the audio streams that are active

Synopsis:

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

int snd_ctl_ducking_read(snd_ctl_t *handle,
                         const char* ducking_output,
                         snd_ducking_status_t **info );
```

Arguments:

handle

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

ducking_output

The names of the ducking output to monitor.

info

A pointer to location where a `snd_ducking_status_t` structure is stored. The structure contains information about the audio types that are active. If the call is successful, the system allocates memory for you, therefore it's the responsibility of the caller to free the allocated memory. This value can't be NULL.

Library:

libasound.so

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

Description:

This function is deprecated. Use *snd_ctl_ducking_status_read()* instead.

The *snd_ctl_ducking_read()* function reads the current ducking status. Use this function to get information about what audio types are active after you receive a SND_CTL_READ_AUDIOMGMT_CHG event in an *audiomgmt* callback.

Returns:

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

Errors:

-EINVAL

Invalid *handle* or *info* argument.

Classification:

QNX Neutrino

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

Related Links

`snd_ducking_status_t` (p. 174)

Information about the audio types that are active in terms of audio concurrency management policies in effect on the system

`snd_ducking_type_status_t` (p. 176)

Information about the active audio type.

snd_ctl_read() (p. 170)

Read pending control events

snd_ctl_ducking_status_read()

Get all active audio subchannels associated with an audio ducking output

Synopsis:

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

int snd_ctl_ducking_status_read( snd_ctl_t *handle,
                                 const char* ducking_output,
                                 snd_ducking_priority_status_t **info );
```

Arguments:

handle

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

ducking_output

The name of the ducking output to monitor. The ducking output is the audio concurrency management context or output specified by the audio configuration file `audiomgmt_id` key.

info

A pointer to location where a `snd_ducking_priority_status_t` structure is stored. The structure contains information about the audio types that are active. If the call is successful, the system allocates memory for you, therefore it's the responsibility of the caller to free the allocated memory. This value can't be NULL.

Library:

libasound.so

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

Description:

The *snd_ctl_ducking_status_read()* function gets a list of all the active audio subchannels and information about their priority and ducking state, regardless of whether they are audible or not.

Because audio concurrency management handles a PCM link group as a single stream, this function reports only information for the first subchannel in a group.

Use this function to get information about active audio types after you receive a SND_CTL_READ_AUDIOMGMT_STATUS_CHG event in an *audiomgmt* callback.

The priority and ducking state information is specified in the `snd_ducking_priority_status_t` structure that *info* points to:

```
typedef struct snd_ducking_priority_status {
    int                       nsubchns;
    snd_ducking_subchn_info_t subchns[];
} snd_ducking_priority_status_t;
```

This structure includes the following members:

nsubchns

The number of active subchannels. Subchannels do not have to be audible to be considered active.

subchns

An array of subchannel ducking information structures.

The subchns member is a variable-sized array of `snd_ducking_status_t` structures:

```
typedef struct snd_ducking_subchn_info
{
    pid_t    pid;
    uint32_t prio;
    uint32_t state;
    uint32_t ducked_by;
    char      name[32];
} snd_ducking_subchn_info_t;
```

The members include:

pid

The process ID of the application that opened the audio stream.

prio

The priority of the audio type. The higher the number, the higher the priority.

state

The ducking state. The following values are valid:

- SND_PCM_DUCKING_STATE_ACTIVE — Ducking is active because the stream is active.
- SND_PCM_DUCKING_STATE_HARD_SUSPENDED — The subchannel is suspended by an audio type with a higher priority.
- SND_PCM_DUCKING_STATE_SOFT_SUSPENDED — The subchannel is suspended by an audio type of the same priority.
- SND_PCM_DUCKING_STATE_PAUSED — The stream will transition to SND_PCM_STATUS_PAUSED when unsuspended.
- SND_PCM_DUCKING_STATE_FORCED_ACTIVE — Ducking was made active using *snd_pcm_channel_audio_ducking()*.

- SND_PCM_DUCKING_STATE_MUTE_BY_HIGHER — The stream was ducked by a stream with a higher priority.
- SND_PCM_DUCKING_STATE_MUTE_BY_SAME — The stream was ducked by a stream with an equal priority.
- SND_PCM_DUCKING_STATE_DUCKED — Ducking is active.

ducked_by

Describes the audio stream that ducked this active stream. Valid values are:

- SND_PCM_DUCKED_BY_SAME_PRIO
- SND_PCM_DUCKED_BY_HIGHER_PRIO
- SND_PCM_DUCKED_BY_NONTRANSIENT

Returns:

EOK on success. The priority status information is allocated and populated. The client application must free the buffer afterwards.

Errors:

-ENOMEM

No memory available to build the ducking status reply.

-EINVAL

Ducking output not found.

Classification:

QNX Neutrino

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

Related Links

snd_ctl_read() (p. 170)
Read pending control events

snd_ctl_set_filter() (p. 172)
Set the control events filter

snd_ctl_get_filter()

Get the current control events filter

Synopsis:

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

int snd_ctl_get_filter( snd_ctl_t *handle, snd_ctl_filter_t * filter );
```

Arguments:

handle

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

filter

A filter that specifies which control events `io-audio` sends to the client application.

Library:

libasound.so

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

Description:

The *snd_ctl_get_filter()* function gets the control event filter as a `snd_ctl_filter_t` structure:

```
typedef struct snd_ctl_filter {
    uint32_t enable;
    uint8_t  reserved[124];
} snd_ctl_filter_t;
```

This structure includes the following members:

enable

A bitfield created using *SND_CTL_EVENT_MASK()*.

reserved

Must be filled with 0 (zero).

Returns:

EOK on success. The filter structure is populated.

Errors:

-EINVAL

Invalid *handle* argument or pointer to filter is NULL.

Classification:

QNX Neutrino

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

Related Links

snd_ctl_open() (p. 164)
Create a connection and handle to the specified control device

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()*.

info

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

Library:

libasound.so

Use the `-l asound` option with `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

Related Links

`snd_ctl_hw_info_t` (p. 156)
Information about a sound card's hardware

snd_ctl_open() (p. 164)
Create a connection and handle to the specified control device

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()*.

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

Related Links

snd_ctl_hw_info() (p. 154)
Get information about a sound card's hardware

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()*.

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 with `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.

Related Links

snd_mixer_group_read() (p. 204)
Get a mixer group's configurable parameters

mix_ctl.c *application sample source code* (p. 499)
This is a sample application that captures the groups and switches in the mixer.

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()*.

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 with `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

Related Links

snd_mixer_groups() (p. 212)

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

mix_ctl.c *application sample source code* (p. 499)

This is a sample application that captures the groups and switches in the mixer.

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()*.

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 with `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.

Related Links

snd_mixer_group_write() (p. 210)
Set a mixer group's configurable parameters

mix_ctl.c *application sample source code* (p. 499)
This is a sample application that captures the groups and switches in the mixer.

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 with `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

Related Links

snd_ctl_close() (p. 143)
Close a control handle

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()*.

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` structure in which *snd_ctl_pcm_channel_info()* stores the information.

Library:

libasound.so

Use the `-l asound` option with `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()* and *snd_pcm_plugin_info()*, 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

Related Links

snd_ctl_open() (p. 164)

Create a connection and handle to the specified control device

snd_pcm_channel_info() (p. 257)

Get information about a PCM channel's current capabilities

`snd_pcm_channel_info_t` (p. 259)

Information structure for a PCM channel

snd_pcm_plugin_info() (p. 376)

Get information about a PCM channel's capabilities (plugin-aware)

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()*.

dev

The PCM device.

info

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

Library:

libasound.so

Use the `-l asound` option with `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

Related Links

snd_ctl_open() (p. 164)
: *Create a connection and handle to the specified control device*

snd_pcm_info() (p. 331)
: *Get general information about a PCM device*

`snd_pcm_info_t` (p. 333)
: *Capability information about a PCM device*

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()*.

callbacks

A pointer to a `snd_ctl_callbacks_t` structure that defines the callbacks for the events.

Library:

libasound.so

Use the `-l asound` option with `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*).

By default, events are disabled and an application must enable them using *snd_ctl_set_filter()*. Use *snd_ctl_set_filter()* and *snd_ctl_get_filter()* to tell `io-audio` which events you are interested in monitoring, handling, or both.

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

Related Links

snd_ctl_callbacks_t (p. 140)
Control callback functions

snd_ctl_file_descriptor() (p. 145)
Get the control file descriptor

snd_ctl_get_filter() (p. 152)
Get the current control events filter

snd_ctl_open() (p. 164)
Create a connection and handle to the specified control device

snd_ctl_set_filter() (p. 172)
Set the control events filter

in the QNX Neutrino *C Library Reference*

snd_ctl_set_filter()

Set the control events filter

Synopsis:

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

int snd_ctl_set_filter( snd_ctl_t *handle, snd_ctl_filter_t * filter );
```

Arguments:

handle

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

filter

A filter that specifies which control events `io-audio` sends to the client application.

Library:

libasound.so

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

Description:

The *snd_ctl_set_filter()* function sets the control event filter using a `snd_ctl_filter_t` structure:

```
typedef struct snd_ctl_filter {
    uint32_t enable;
    uint8_t  reserved[124];
} snd_ctl_filter_t;
```

This structure includes the following members:

enable

A bitfield that represents a control event type. Use the *SND_CTL_EVENT_MASK()* macro to translate one of the following control event types to the filter bitmask:

- SND_CTL_READ_REBUILD
- SND_CTL_READ_SWITCH_VALUE
- SND_CTL_READ_SWITCH_ADD
- SND_CTL_READ_SWITCH_REMOVE
- SND_CTL_READ_AFM_STATE
- SND_CTL_READ_AFM_AUDIO_MODE
- SND_CTL_READ_AFM_DATASET

- SND_CTL_READ_AFM_DETECT
- SND_CTL_READ_AUDIOMGMT_STATUS_CHG

reserved

Must be filled with 0 (zero).

Returns:

Zero on success, or a negative value on error.

Errors:

-EINVAL

Invalid *handle* argument or pointer to filter is NULL.

Classification:

QNX Neutrino

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

Related Links

snd_ctl_open() (p. 164)
Create a connection and handle to the specified control device

snd_ducking_status_t

Information about the audio types that are active in terms of audio concurrency management policies in effect on the system

Synopsis:

```
typedef struct snd_ducking_output_status {
    int       ntypes;
    snd_ducking_type_status_t active_types[1];
    #define SND_DUCKING_STATUS_NEXT_TYPE(type) \
                  ((void*)type + sizeof(*type) + \
                  (sizeof(type->pids[0]) * (type->npids - 1)))
} snd_ducking_status_t;
```

Description:

The list of active audio types currently running on the system and the information associated with each audio type. This datatype defines a *SND_DUCKING_STATUS_NEXT_TYPE*(type) macro, which you use to traverse the next structure. To use it, you pass a pointer to *snd_ducking_type_status_t* that you retrieve from the *info* member from calling *snd_ctl_ducking_read()*. Here's an example of a callback that uses the macro:

```
void audiomgmt_cb ( snd_ctl_t *hdl, void *private_data, int cmd)
{
 int status, i, j;
 const char *ducking_output = private_data;
 snd_ducking_status_t *info = NULL;
 snd_ducking_type_status_t *active_type = NULL;


 switch (cmd)
 {
  case SND_CTL_READ_AUDIOMGMT_CHG:
   if ((status = snd_ctl_ducking_read(hdl, ducking_output, &info )) != EOK)
   {
    printf("Failed to read ducking info - %s\n", snd_strerror(status));
    return;
   }

   printf("\nActive audio types = %d\n", info->ntypes);
   /* The active_types structure is variable in size and is based on the
    * number of PIDs it references. For this reason, you must
    * walk the active_type member of the info structure using the
    * SND_DUCKING_STATUS_NEXT_TYPE() macro.
    */
   for (i = 0, active_type = info->active_types;
                i < info->ntypes;
                i++, active_type = SND_DUCKING_STATUS_NEXT_TYPE(active_type))
   {
    printf("Audio Type %s, Priority %d\n", active_type->name, active_type->prio);
    printf("\tPids (%d): ", active_type->npids);
    for (j = 0; j  < active_type->npids; j++)
    {
     printf("%d ", active_type->pids[j]);
    }
    printf("\n");
   }
   /* snd_ctl_ducking_read() allocates memory for the info buffer,
    * so you must free the memory that was allocated for you.
    */
   free(info);
   break;
  default:
   break;
 }
}
```

The members include:

ntypes

The number of active audio types.

active_types

An array of active audio types.

Classification:

QNX Neutrino

Related Links

snd_ctl_ducking_read() (p. 147)

Get the audio streams that are active

snd_ducking_type_status_t

Information about the active audio type.

Synopsis:

```
typedef struct snd_ducking_type_status {
    char name[32];
    int prio;
    int npids;
    pid_t pids[1];
} snd_ducking_type_status_t;
```

Description:

Provides information about the active audio type.

The members include:

name

The name of the audio type.

prio

The priority of the audio type. The higher the number, the higher the priority.

npids

The number of process with active audio streams with this audio type.

pids

An array of process IDs representing the active audio streams with this audio type.

Classification:

QNX Neutrino

Related Links

snd_ctl_ducking_read() (p. 147)
Get the audio streams that are active

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()*. 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` 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` structure that holds the ID of the group affected by the event.

Classification:

QNX Neutrino

Related Links

`snd_mixer_eid_t` (p. 181)
Mixer element ID structure

`snd_mixer_gid_t` (p. 202)
Mixer group ID structure

snd_mixer_read() (p. 223)
Read pending mixer events

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()*.

Library:

libasound.so

Use the `-l asound` option with `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

Related Links

snd_mixer_open() (p. 219)

Create a connection and handle to a specified mixer device

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

Related Links

`snd_mixer_element_t` (p. 185)
Mixer element control structure

snd_mixer_elements() (p. 189)
Get the number of elements in the mixer and their element IDs

`snd_mixer_group_t` (p. 206)
Mixer group control structure

snd_mixer_read() (p. 223)

  Read pending mixer events

snd_mixer_routes() (p. 225)

  Get the number of routes in the mixer and their IDs

snd_mixer_sort_eid_table() (p. 233)

  Sort a list of element ID structures

`snd_pcm_channel_info_t` (p. 259)

  Information structure for a PCM channel

snd_pcm_channel_setup() (p. 279)

  Get the current configuration for the specified PCM channel

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()*.

element

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

Library:

libasound.so

Use the `-l asound` option with `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.

Related Links

`snd_mixer_element_t` (p. 185)
: *Mixer element control structure*

snd_mixer_element_write() (p. 187)
: *Set a mixer element's configurable parameters*

snd_mixer_elements() (p. 189)
: *Get the number of elements in the mixer and their element IDs*

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_balance1          balance1;
        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

Related Links

snd_mixer_eid_t (p. 181)
 Mixer element ID structure

snd_mixer_element_read() (p. 183)

Get a mixer element's configurable parameters

snd_mixer_element_write() (p. 187)

Set a mixer element's configurable parameters

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()*.

element

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

Library:

libasound.so

Use the `-l asound` option with `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()*.

Related Links

snd_mixer_element_read() (p. 183)
Get a mixer element's configurable parameters

`snd_mixer_element_t` (p. 185)
Mixer element control structure

snd_mixer_elements() (p. 189)
Get the number of elements in the mixer and their element IDs

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()*.

elements

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

Library:

libasound.so

Use the `-l asound` option with `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` 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

Related Links

`snd_mixer_eid_t` (p. 181)
Mixer element ID structure

snd_mixer_element_read() (p. 183)
Get a mixer element's configurable parameters

snd_mixer_element_write() (p. 187)
Set a mixer element's configurable parameters

`snd_mixer_elements_t` (p. 191)
Information about all elements in a mixer

snd_mixer_sort_eid_table() (p. 233)
Sort a list of element ID structures

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()*.

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`* structures.

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

Classification:

QNX Neutrino

Related Links

`snd_mixer_eid_t` (p. 181)
Mixer element ID structure

snd_mixer_elements() (p. 189)
Get the number of elements in the mixer and their element IDs

snd_mixer_element_volume1_range_t

Mixer minimum

Synopsis:

```
typedef struct snd_mixer_element_volume1_range
{
 int32_t  min, max;
 int32_t  min_dB, max_dB;
 int32_t  dB_scale_factor;
 uint8_t  reserved[124];
}snd_mixer_element_volume1_range_t;
```

Description:

The `snd_mixer_element_volume1_range` structure describes the range of the volume element. The members include:

min

The minimum volume increments.

maxium

The maximum volume increments.

min_dB

The minimum decibel (dB) range value that's multiplied by the *dB_scale_factor*.

min_dB

The maximum decibel (dB) range value that's multiplied by the *dB_scale_factor*.

db_scale_factor

The value by which *min_dB* and *max_dB* have been scaled.

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()*.

Library:

libasound.so

Use the `-l asound` option with `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()*.

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

Safety:	
Signal handler	Yes
Thread	Yes

Related Links

snd_mixer_read() (p. 223)
Read pending mixer events

in the QNX Neutrino *C Library Reference*

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()* to specify the events you want to track, and *snd_mixer_get_filter()* 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

Related Links

snd_mixer_get_filter() (p. 200)

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

snd_mixer_set_filter() (p. 231)

Set the mask of mixer events that the driver will track

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 with `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
Thread	Yes

Related Links

snd_mixer_set_bit() (p. 229)

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

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()*.

filter

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

Library:

libasound.so

Use the `-l asound` option with `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()*. You can use *snd_mixer_read()* 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

Related Links

`snd_mixer_filter_t` (p. 196)
Information about a mixer's filters

snd_mixer_read() (p. 223)
Read pending mixer events

snd_mixer_set_filter() (p. 231)
Set the mask of mixer events that the driver will track

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

Related Links

snd_mixer_group_read() (p. 204)
Get a mixer group's configurable parameters

`snd_mixer_group_t` (p. 206)
Mixer group control structure

snd_mixer_groups() (p. 212)
Get the number of groups in the mixer and their group IDs

snd_mixer_sort_gid_table() (p. 235)
Sort a list of group ID structures

`snd_pcm_channel_info_t` (p. 259)
: *Information structure for a PCM channel*

snd_pcm_channel_setup() (p. 279)
: *Get the current configuration for the specified PCM channel*

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()*.

group

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

Library:

libasound.so

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

Description:

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

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.

Related Links

`snd_mixer_gid_t` (p. 202)
: *Mixer group ID structure*

`snd_mixer_group_t` (p. 206)
: *Mixer group control structure*

snd_mixer_group_write() (p. 210)
: *Set a mixer group's configurable parameters*

snd_mixer_groups() (p. 212)
: *Get the number of groups in the mixer and their group IDs*

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;
    int32_t     dB_scale_factor;
    uint32_t    balance_min;
    uint32_t    balance_max;
    uint32_t    balance_level;
    uint32_t    fade_level;
    uint8_t     reserved[99];       /* must be filled with zero */
}       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()*, and set it by calling *snd_mixer_group_write()*.

Some structure members are read-only and are ignored for *snd_mixer_group_write()* calls. The *change_duration* member is write-only and ignored for *snd_mixer_group_read()* calls.

The members of this structure include:

gid

A *snd_mixer_gid_t* 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. Read only:

- 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_BALANCE — group has balance controls.
- SND_MIXER_GRPCAP_FADE — group has fade controls.
- 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. Read only.

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. Read only.

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`. Read only.

elements

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

elements_over

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

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`* 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. Write only.

min_dB*, *max_dB

The minimum and maximum sound levels, in decibels. Read only.

dB_scale_factor

The value by which *min_dB* and *max_dB* have been scaled. Read only.

balance_min

The minimum balance. Read only. See *"Balance and fade controls."*

balance_max

The maximum balance. Read only.

balance_level

The balance level.

fade_level

The fade level.

Balance and fade controls

The balance and fade controls work like a slider where 0% corresponds to the left (balance) or front (fade) channel and 100% to the right (balance) or rear (fade) channel. At 50%, the balance or fade control applies no attenuation to any channels. A value less than 50% is closer to left or front and attenuates right or rear, and a value greater than 50% is closer to right or rear and attenuates left or front. For the center channel, moving the balance away from 50% in either direction causes an attenuation. The woofer and low-frequency effects (LFE) channels are treated like a rear channel — they are attenuated when you fade towards the front.

The *balance_min* member corresponds to a 0% value for the range and *balance_max* corresponds to a 100% value. For example, if *balance_min* is 0 and *balance_max* is 200, a *balance_level* of 100 balances the volume.

Classification:

QNX Neutrino

Related Links

`snd_mixer_eid_t` (p. 181)
Mixer element ID structure

`snd_mixer_gid_t` (p. 202)
Mixer group ID structure

snd_mixer_group_read() (p. 204)
Get a mixer group's configurable parameters

snd_mixer_group_write() (p. 210)
Set a mixer group's configurable parameters

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()*.

group

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

Library:

libasound.so

Use the `-l asound` option with `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()*.

Related Links

snd_mixer_group_read() (p. 204)
 Get a mixer group's configurable parameters

`snd_mixer_group_t` (p. 206)
 Mixer group control structure

snd_mixer_groups() (p. 212)
 Get the number of groups in the mixer and their group IDs

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()*.

groups

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

Library:

libasound.so

Use the `-l asound` option with `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` 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

Related Links

`snd_mixer_gid_t` (p. 202)
Mixer group ID structure

snd_mixer_group_read() (p. 204)
Get a mixer group's configurable parameters

snd_mixer_group_write() (p. 210)
Set a mixer group's configurable parameters

`snd_mixer_groups_t` (p. 214)
Information about all of the mixer groups

snd_mixer_sort_gid_table() (p. 235)
Sort a list of group ID structures

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()*.

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` structures.

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

Classification:

QNX Neutrino

Related Links

snd_mixer_groups() (p. 212)
Get the number of groups in the mixer and their group IDs

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()*.

info

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

Library:

libasound.so

Use the `-l asound` option with `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

Related Links

`snd_mixer_info_t` (p. 217)
Information about a mixer

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()*.

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

Related Links

snd_mixer_info() (p. 215)

Get general information about a mixer device

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 with `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

Related Links

snd_mixer_close() (p. 179)
Close a mixer handle

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 with `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

Related Links

snd_mixer_close() (p. 179)
Close a mixer handle

snd_mixer_open() (p. 219)
Create a connection and handle to a specified mixer device

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()*.

callbacks

A pointer to a `snd_mixer_callbacks_t` structure that defines the list of callbacks.

Library:

libasound.so

Use the `-l asound` option with `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

Related Links

`snd_mixer_callbacks_t` (p. 177)
List of mixer callback functions

`snd_mixer_eid_t` (p. 181)
Mixer element ID structure

snd_mixer_file_descriptor() (p. 194)
Return the file descriptor of the connection to the sound mixer interface

snd_mixer_get_filter() (p. 200)
Get the current mask of mixer events that the driver is tracking

snd_mixer_set_filter() (p. 231)
Set the mask of mixer events that the driver will track

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()*.

routes

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

Library:

libasound.so

Use the `-l asound` option with `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` 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

Related Links

`snd_mixer_eid_t` (p. 181)
Mixer element ID structure

snd_mixer_elements() (p. 189)
Get the number of elements in the mixer and their element IDs

snd_mixer_groups() (p. 212)
Get the number of groups in the mixer and their group IDs

`snd_mixer_routes_t` (p. 227)
Information about mixer routes

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()*.

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`* 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`* structures.

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

Classification:

QNX Neutrino

Related Links

snd_mixer_routes() (p. 225)
Get the number of routes in the mixer and their IDs

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 with `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
Signal handler	Yes
Thread	Yes

Related Links

snd_mixer_get_bit() (p. 198)

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

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()*.

filter

A pointer to a `snd_mixer_filter_t` structure that defines a mask of events to track.

Library:

libasound.so

Use the `-l asound` option with `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()*. You can use *snd_mixer_read()* 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

Related Links

snd_mixer_file_descriptor() (p. 194)
: *Return the file descriptor of the connection to the sound mixer interface*

`snd_mixer_filter_t` (p. 196)
: *Information about a mixer's filters*

snd_mixer_get_filter() (p. 200)
: *Get the current mask of mixer events that the driver is tracking*

snd_mixer_read() (p. 223)
: *Read pending mixer events*

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`, 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` 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 with `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

Related Links

`snd_mixer_eid_t` (p. 181)
Mixer element ID structure

snd_mixer_elements() (p. 189)
Get the number of elements in the mixer and their element IDs

`snd_mixer_weight_entry_t` (p. 237)
Weight table for sorting mixer element and group IDs

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* 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* 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 with `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
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

Related Links

`snd_mixer_gid_t` (p. 202)
Mixer group ID structure

snd_mixer_groups() (p. 212)
Get the number of groups in the mixer and their group IDs

`snd_mixer_weight_entry_t` (p. 237)
Weight table for sorting mixer element and group IDs

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()* and *snd_mixer_sort_gid_table()* 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

Related Links

snd_mixer_sort_eid_table() (p. 233)
Sort a list of element ID structures

snd_mixer_sort_gid_table() (p. 235)
Sort a list of group ID structures

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 with `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()*.

Returns:

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

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:

Signal handler	Yes
Thread	Yes

Related Links

snd_pcm_format_big_endian() (p. 302)
: *Check for a big-endian format*

snd_pcm_format_linear() (p. 304)
: *Check for a linear format*

snd_pcm_format_little_endian() (p. 306)
: *Check for a little-endian format*

snd_pcm_format_signed() (p. 308)
: *Check for a signed format*

snd_pcm_format_size() (p. 310)
: *Convert the size in the given samples to bytes*

snd_pcm_format_unsigned() (p. 314)
: *Check for an unsigned format*

snd_pcm_format_width() (p. 316)
: *Return the sample width in bits for a format*

snd_pcm_get_format_name() (p. 327)
: *Convert a format value into a human-readable text string*

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EBADFD

The pcm device state isn't ready.

-EINTR

The operation was interrupted because of a system signal (such as a timer) or an error was returned asynchronously.

-EINVAL

The state of handle is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_channel_flush() (p. 252)
Flush all pending data in a PCM channel's queue and stop the channel

snd_pcm_playback_drain() (p. 358)
Stop the PCM playback channel and discard the contents of its queue

snd_pcm_playback_flush() (p. 360)
Play out all pending data in a PCM playback channel's queue and stop the channel

snd_pcm_plugin_flush() (p. 371)
Finish processing all pending data in a PCM channel's queue and stop the channel

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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()*). 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()*.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid, an invalid file descriptor to the channel for capture, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_channel_go() (p. 255)
  Start a PCM channel running

snd_pcm_playback_go() (p. 363)
  Start a PCM playback channel running

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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 on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid, and invalid *channel* was provided isn't capturing, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-ENOTSUP

Pause isn't supported on the PCM device that's referenced by the PCM handle (*pcm*).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_resume() (p. 248)
  Resume a channel that was paused while capturing

snd_pcm_open() (p. 347)
  Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
  Create a handle and open a connection to the preferred audio interface

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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()*) 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:

EOK on success, negative error code, or a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Listed below is additional information for common error values:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-EBUSY

The specified channel is running.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_channel_go() (p. 255)
Start a PCM channel running

snd_pcm_channel_prepare() (p. 272)
Signal the driver to ready the specified channel

snd_pcm_playback_prepare() (p. 367)
Signal the driver to ready the playback channel

snd_pcm_plugin_prepare() (p. 383)
Signal the driver to ready the specified channel (plugin-aware)

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

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

Description:

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

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of the handle is invalid, the channel wasn't being used for capturing, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-ENOTSUP

Resume isn't supported on the PCM device that's referenced by the PCM handle (*pcm*).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_pause() (p. 244)
: *Pause a channel that's capturing*

snd_pcm_open() (p. 347)
: *Create a handle and open a connection to a specified audio interface*

snd_pcm_open_preferred() (p. 353)
: *Create a handle and open a connection to the preferred audio interface*

snd_pcm_channel_audio_ducking()

Enable or disable audio ducking

Synopsis:

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

int snd_pcm_channel_audio_ducking( snd_pcm_t *pcm,
                                   int channel,
                                   uint32_t enable );
```

Arguments:

pcm

A handle to the PCM device, which you must have opened by calling *snd_pcm_open_name()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel direction. The only valid value is SND_PCM_CHANNEL_PLAYBACK.

enable

An integer value that specifies whether to enable or disable audio ducking on the specified channel. A zero indicates to disable audio ducking while a non-zero values indicates to enable audio ducking. Once enabled, the channel conforms to the behavior specified by the audio ducking algorithms enabled on the system.

Library:

libasound.so

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

Description:

The *snd_pcm_channel_audio_ducking()* function permits audio ducking to occur for the specified *channel* (audio stream), regardless of its streaming state. By default, an audio stream is part of the audio policy algorithm that's described in the *Audio Concurrency Management* chapter of this guide. If audio ducking isn't configured on the system, then this function doesn't have an effect on the specified audio channel.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

CAUTION: If no audio policy configuration file is provided, this function returns EOK.

Errors:

-EINVAL

The handle (*pcm*) is invalid, the specified channel (*channel*) has an invalid file descriptor, or the channel direction (*channel*) is invalid (e.g., *SND_PCM_CHANNEL_CAPTURE* isn't a valid channel direction for this function). This error also occurs if the channel is in the SND_PCM_STATUS_NOTREADY state.

-ENOTSUP

If the call is targeted against a PCM interface that doesn't have the PCM software mixer enabled. The mixer is required to perform audio concurrency management.

Classification:

QNX Neutrino

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

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

Library:

libasound.so

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

Description:

The *snd_pcm_channel_flush()* function flushes all unprocessed data in the driver queue by calling *snd_pcm_capture_flush()* or *snd_pcm_playback_flush()*, depending on the value of *channel*. These are the considerations:

- If the channel is processing playback data, the call blocks until all data in the driver queue is played out the channel.
- If the channel is processing capture data, any unread data in the driver queue is discarded.

If you use this call to process playback data, there are two considerations:

- Since this call is a blocking call, if the channel is in the SND_PCM_STATUS_SUSPENDED state, an error occurs if you call this function.
- If this function is called while in the SND_PCM_STATUS_PAUSED state and during playback data processing, playback is resumed, the channel goes into the SND_PCM_STATUS_RUNNING state immediately, and then ends in the SND_PCM_STATUS_READY state.

  If the channel doesn't start playing (not enough data has been written to start playback), then this call silences the remainder of the buffer to satisfy the playback start requirements. Once the playback requirements are satisfied, playback starts and allows the data to be played out. Once the buffer empties, this call returns successfully.

If this call completes successfully, the channel is left in the SND_PCM_STATUS_READY state.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EAGAIN

The state was SND_PCM_STATUS_SUSPENDED while trying to call this function. This error is also returned if the state was SND_PCM_STATUS_PAUSED, but resuming the playback failed.

-EBADFD

The PCM device state isn't Ready.

-EINTR

The operation was interrupted because of a system signal (such as a timer) or an error was returned asynchronously.

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_flush() (p. 240)
    Discard all pending data in a PCM capture channel's queue and stop the channel

snd_pcm_playback_drain() (p. 358)
    Stop the PCM playback channel and discard the contents of its queue

snd_pcm_playback_flush() (p. 360)
    Play out all pending data in a PCM playback channel's queue and stop the channel

snd_pcm_plugin_flush() (p. 371)
    Finish processing all pending data in a PCM channel's queue and stop the channel

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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()* or *snd_pcm_playback_go()*, 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()*). 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()*.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_go() (p. 242)
Start a PCM capture channel running

snd_pcm_playback_go() (p. 363)
Start a PCM playback channel running

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

info

A pointer to a `snd_pcm_channel_info_t` 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 with `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()*, 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()*, which gets information about the *complete* capabilities of the system.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid, an invalid *params* was provided as input, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid and if you want to recover from the error.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_ctl_pcm_channel_info() (p. 166)
Get information about a PCM channel's capabilities from a control handle

`snd_pcm_channel_info_t` (p. 259)
Information structure for a PCM channel

snd_pcm_plugin_info() (p. 376)
Get information about a PCM channel's capabilities (plugin-aware)

snd_pcm_channel_info_t

Information structure for a PCM channel

Synopsis:

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

typedef struct snd_pcm_channel_info
{
    int32_t     subdevice;
    int8_t      subname[36];
    int32_t     channel;
    int32_t     zero1;
    int32_t     output_class;
    int32_t     output_classes;
    int32_t     zero2[2];
    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.

zerp1

Padding of size uint32_t.

output_class

The output class that's used for DRM purposes. These are the valid values you can use:

SND_OUTPUT_CLASS_UKNOWN

The output channel is for an unknown type.

SND_OUTPUT_CLASS_SPEAKER

Indicates that the output channel is for speakers connected to the system.

SND_OUTPUT_CLASS_HEADPHONE

Indicates that the output channel is for headphones connected to the system.

SND_OUTPUT_CLASS_LINEOUT

Indicates that the output channel is for the line-out for the system.

SND_OUTPUT_CLASS_BLUETOOTH

Indicates that the output channel is for Bluetooth.

SND_OUTPUT_CLASS_TOSLINK

Indicates the output channel is for an S-Link connector.

SND_OUTPUT_CLASS_MIRACAST

Indicates that the output channel is for Miracast.

SND_NUM_OUTPUT_CLASSES

An end-of-list identifier that indicates the total number of output types recognized by this library.

output_classes

The mask of the supported output classes that are used for DRM purposes.

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 PCM device, underlying hardware, or both supports pausing and resuming of the audio stream (playback only).
- SND_PCM_CHNINFO_RESTRICTED — the hardware supports muting of channel data.
- SND_PCM_CHNINFO_ROUTING — the hardware supports redirecting the channel to a different transducer.
- SND_PCM_CHNINFO_LOGGING — PCM logging is enabled on the channel.
- SND_PCM_CHNINFO_PROTECTED_VOICE — the hardware supports a protected voice channel. A protected voice channel contains only voice data.

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` structure that describes the mixer element identification for this channel.

mixer_gid

The mixer group identification for this channel; see `snd_mixer_gid_t`. 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()*. 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()*), this mixer group is the subchannel mixer group that's specific to the application's current subchannel.

Classification:

QNX Neutrino

Related Links

snd_ctl_pcm_channel_info() (p. 166)
Get information about a PCM channel's capabilities from a control handle

`snd_mixer_eid_t` (p. 181)
: *Mixer element ID structure*

`snd_mixer_gid_t` (p. 202)
: *Mixer group ID structure*

snd_pcm_channel_info() (p. 257)
: *Get information about a PCM channel's current capabilities*

snd_pcm_plugin_info() (p. 376)
: *Get information about a PCM channel's capabilities (plugin-aware)*

Audio Policy Management (p. 63)
: You can manage multiple audio streams using various audio policies.

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

params

A pointer to a `snd_pcm_channel_params_t` 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 with `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()*, unless you've disabled the conversion by calling:

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

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid, an invalid *params* was provided as input, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid and if you want to recover from the error.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

`snd_pcm_channel_params_t` (p. 266)
PCM channel parameters

snd_pcm_channel_setup() (p. 279)
Get the current configuration for the specified PCM channel

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

snd_pcm_plugin_params() (p. 378)
Set the configurable parameters for a PCM channel (plugin-aware)

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;
    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];
    char        audio_type_name[32];
    uint8_t     reserved[64];            /* 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()* 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.

sync

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

format

The data format; see *snd_pcm_format_t*.

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_ROLLOVER_RESET — when recovering from writing or reading more data, reset the counter. Use this option for fault tolerant systems.
- SND_PCM_STOP_ROLLOVER — Rollover the fragments (i.e., automatically reprepare and continue) when an underrun or overrun occurs. Use this option to overcome jitter between audio that is being captured or played by the client application.

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.

why_failed

An indication why an operation failed; one of:

- SND_PCM_PARAMS_BAD_MODE
- SND_PCM_PARAMS_BAD_START
- SND_PCM_PARAMS_BAD_STOP
- SND_PCM_PARAMS_BAD_FORMAT
- SND_PCM_PARAMS_BAD_RATE
- SND_PCM_PARAMS_BAD_VOICES
- SND_PCM_PARAMS_NO_CHANNEL

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.

audio_type_name

The name of the audio type the subchannel is associated with. This is used for audio concurrency management policies, such as audio ducking. For more information, see the *Audio Concurrency Management* chapter. If there's no matching audio type defined in your audio policy configuration file, then the audio type named `default` is used. If there's no audio type named `default`, the lowest priority audio type is used instead.

Classification:

QNX Neutrino

Related Links

snd_pcm_channel_params() (p. 264)
Set a PCM channel's configurable parameters

`snd_pcm_format_t` (p. 312)
PCM data format structure

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

Library:

libasound.so

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

Description:

The *snd_pcm_channel_pause()* function pauses a channel by calling *snd_pcm_capture_pause()* or *snd_pcm_playback_pause()*, 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 on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-EIO

The channel isn't valid that was passed in was not set to SND_PCM_CHANNEL_PLAYBACK or SND_PCM_CHANNEL_CAPTURE.

-ENOTSUP

Pause isn't supported on the PCM device that's referenced by the PCM handle (*pcm*).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_pause() (p. 244)
Pause a channel that's capturing

snd_pcm_channel_resume() (p. 276)
Resume a channel

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

snd_pcm_playback_pause() (p. 365)
Pause a channel that's playing back

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

Library:

libasound.so

Use the `-l asound` option with `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()* or *snd_pcm_playback_prepare()*, 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()*) 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:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-EBUSY

Channel is running.

-EIO

Channel was not set to SND_PCM_CHANNEL_PLAYBACK or SND_PCM_CHANNEL_CAPTURE.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_prepare() (p. 246)
Signal the driver to ready the capture channel

snd_pcm_channel_go() (p. 255)
Start a PCM channel running

snd_pcm_playback_prepare() (p. 367)
Signal the driver to ready the playback channel

snd_pcm_plugin_prepare() (p. 383)
Signal the driver to ready the specified channel (plugin-aware)

snd_pcm_channel_read_event()

Get a PCM event

Synopsis:

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

int snd_pcm_channel_read_event (snd_pcm_t * pcm,
                                int channel,
                                snd_pcm_event_t * event);
```

Arguments:

pcm

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

channel

The channel to retrieve an event for; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

event

A pointer to a `snd_pcm_event_t` in which *snd_pcm_channel_read_event()* retrieve to get information about the event.

Library:

libasound.so

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

Description:

This function retrieves the PCM events (`snd_pcm_event_t`) for the specified channel. To receive events, the client application must first register for event using *snd_pcm_set_filter()*. This call is a non-blocking call. Use *select()* (with `excepttfds`) or *poll* (with POLLRDBAND).

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of the *pcm* or *event* is invalid or the specified channel doesn't exist.

-EGAIN

There weren't any events available. You can call the function again.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_get_filter() (p. 325)
Retrieves the PCM filters that your application is subscribed to.

snd_pcm_set_filter() (p. 427)
Enable or disable the generation of PCM events

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

Library:

libasound.so

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

Description:

The *snd_pcm_channel_resume()* function resumes a channel by calling *snd_pcm_capture_resume()* or *snd_pcm_playback_resume()*, depending on the value of *channel*.

If the channel is in the SUSPENDED (SND_PCM_STATUS_SUSPENDED), one of the following occurs when you call this function:

- If the channel is hard suspended, calling this function clears any underlying paused condition (e.g., *snd_pcm_channel_pause()* was called before moving to the SND_PCM_STATUS_SUSPENDED state). However, because you are in the SND_PCM_STATUS_SUSPENDED state and waiting for higher priority audio stream to complete, the channel remains in the SUSPENDED state.
- If the channel is in the soft suspended state, calling this function clears any underlying paused condition (e.g., *snd_pcm_channel_pause()* was called before moving to SND_PCM_STATUS_SUSPENDED state) and the channel plays immediately and transitions to the SND_PCM_STATUS_RUNNING state.
- If both the hard suspended and soft suspended conditions occur, the hard suspended state takes precedence. In other words, calling the function clears any pending paused conditione.g., *snd_pcm_channel_pause()*) but the channel remains in the SND_PCM_STATUS_SUSPENDED state.

For more information, see the *Audio Concurrency Management* chapter in this guide.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-EIO

The channel isn't valid that was passed in was not set to SND_PCM_CHANNEL_PLAYBACK or SND_PCM_CHANNEL_CAPTURE.

-ENOTSUP

Resume isn't supported on the PCM device that's referenced by the PCM handle (*pcm*).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_resume() (p. 248)
Resume a channel that was paused while capturing

snd_pcm_channel_pause() (p. 270)
Pause a channel

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

snd_pcm_playback_resume() (p. 369)
    Resume a channel that was paused while playing back

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

setup

A pointer to a `snd_pcm_channel_setup_t` 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 with `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()*, 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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_channel_params() (p. 264)
: *Set a PCM channel's configurable parameters*

`snd_pcm_channel_setup_t` (p. 281)
: *Current configuration of a PCM channel*

`snd_mixer_gid_t` (p. 202)
: *Mixer group ID structure*

snd_pcm_open() (p. 347)
: *Create a handle and open a connection to a specified audio interface*

snd_pcm_open_preferred() (p. 353)
: *Create a handle and open a connection to the preferred audio interface*

snd_pcm_plugin_setup() (p. 401)
: *Get the current configuration for the specified PCM channel (plugin aware)*

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*. 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). When asynchronous sample rate conversion (ASRC) is enabled, this value can shrink or grow while the audio stream is active. When ASRC is disabled, this value is the same as *maximum_frag_size*.

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. When asynchronous sample rate conversion (ASRC) is enabled, *frag_size* can shrink or grow while the audio stream is active and this value represents the maximum fragment size. When ASRC is disabled, *max_frag_size* is equal to the value of *frag_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*.

mmap_valid

The channel can use mmapped access.

mmap_active

The channel is using mmapped transfers.

mixer_card

The mixer card.

Classification:

QNX Neutrino

Related Links

`snd_mixer_gid_t` (p. 202)
Mixer group ID structure

snd_pcm_channel_setup() (p. 279)
Get the current configuration for the specified PCM channel

`snd_pcm_format_t` (p. 312)
PCM data format structure

snd_pcm_plugin_setup() (p. 401)
Get the current configuration for the specified PCM channel (plugin aware)

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

status

A pointer to a `snd_pcm_channel_status_t` 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 with `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()*, 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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

`snd_pcm_channel_status_t` (p. 286)
PCM channel status structure

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

snd_pcm_plugin_status() (p. 403)
Get the runtime status of a PCM channel (plugin aware)

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;
    uint32_t                    ducking_state;
    snd_pcm_status_data_t       status_data
    struct timeval              stop_time;
    uint8_t                     reserved[112];  /* must be filled with zeros */
} 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.

ducking_state

The audio concurrency management ducking state. These are the valid states:

- SND_PCM_DUCKING_STATE_INACTIVE—Ducking isn't active.
- SND_PCM_DUCKING_STATE_ACTIVE—Ducking is active due because the stream is active.
- SND_PCM_DUCKING_STATE_HARD_SUSPENDED —The subchannel is hard-suspended and can't transition from the SUSPENDED state if you call *snd_pcm_*_resume()*.

- SND_PCM_DUCKING_STATE_SOFT_SUSPENDED—The subchannel is soft-suspended and an can transition from the SUSPENDED state if you call *snd_pcm_*_resume()*.
- SND_PCM_DUCKING_STATE_PAUSED—The subchannel will transition to the PAUSED state from the SUSPENDED state.
- SND_PCM_DUCKING_STATE_FORCED_ACTIVE—Ducking is active because the *snd_pcm_channel_audio_ducking()* was called.
- SND_PCM_DUCKING_STATE_MUTE_BY_HIGHER—Ducking occurred because a higher priority subchannel has started running.
- SND_PCM_DUCKING_STATE_MUTE_BY_SAME—Ducking occurred because a same priority subchannel has started running.

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()*, 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()*, *snd_pcm_capture_prepare()*, *snd_pcm_playback_prepare()*, or *snd_pcm_plugin_prepare()*, the state is changed to SND_PCM_STATUS_PREPARED.
- SND_PCM_STATUS_PREPARED — the driver is prepared for operation.
- SND_PCM_STATUS_RUNNING — the driver is actively transferring data through the hardware.
- SND_PCM_STATUS_PAUSED — the channel has stopped playing.
- SND_PCM_STATUS_SUSPENDED — the driver has stopped playing.
- 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()*, *snd_pcm_playback_prepare()*, or *snd_pcm_plugin_prepare()*. 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()*, *snd_pcm_capture_prepare()*, or *snd_pcm_plugin_prepare()*. 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* 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* 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.

- 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()*.
- 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_chmap_query_t

Entry in an array of channel maps

Synopsis:

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

typedef struct snd_pcm_chmap_query {
        enum snd_pcm_chmap_type type;
        snd_pcm_chmap_t map;
} snd_pcm_chmap_query_t;
```

Description:

The `snd_pcm_chmap_query_t` structure describes a channel map. It's used as the entries in an array returned by *snd_pcm_query_chmaps()*. The members include:

type

The type of the channel map; one of the following:

- SND_CHMAP_TYPE_NONE — unspecified channel position
- SND_CHMAP_TYPE_FIXED — fixed channel position
- SND_CHMAP_TYPE_VAR — freely swappable channel position
- SND_CHMAP_TYPE_PAIRED — pair-wise swappable channel position

map

A pointer to a `snd_pcm_chmap_t` structure that describes the channel map.

Classification:

QNX Neutrino

Related Links

`snd_pcm_chmap_t` (p. 291)
Information about a channel map

snd_pcm_free_chmaps() (p. 318)
Free a list of channel mappings

snd_pcm_query_chmaps() (p. 413)
Get a list of the available channel mappings for a PCM stream

snd_pcm_chmap_t

Information about a channel map

Synopsis:

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

typedef struct snd_pcm_chmap {
        unsigned int channels;
        unsigned int pos[0];
} snd_pcm_chmap_t;
```

Description:

The `snd_pcm_chmap_t` structure describes a channel map. The members include:

channels

The number of channels.

pos

An array of channel positions. Each entry is one of the following:

- SND_CHMAP_UNKNOWN — unspecified
- SND_CHMAP_NA — N/A, silent
- SND_CHMAP_MONO — mono stream
- SND_CHMAP_FL — front left
- SND_CHMAP_FR — front right
- SND_CHMAP_RL — rear left
- SND_CHMAP_RR — rear right
- SND_CHMAP_FC — front center
- SND_CHMAP_LFE — low-frequency effects (LFE)
- SND_CHMAP_SL — side left
- SND_CHMAP_SR — side right
- SND_CHMAP_RC — rear center
- SND_CHMAP_FLC — front left center
- SND_CHMAP_FRC — front right center
- SND_CHMAP_RLC — rear left center
- SND_CHMAP_RRC — rear right center
- SND_CHMAP_FLW — front left wide
- SND_CHMAP_FRW — front right wide
- SND_CHMAP_FLH — front left high
- SND_CHMAP_FCH — front center high

- SND_CHMAP_FRH — front right high
- SND_CHMAP_TC — top center
- SND_CHMAP_TFL — top front left
- SND_CHMAP_TFR — top front right
- SND_CHMAP_TFC — top front center
- SND_CHMAP_TRL — top rear left
- SND_CHMAP_TRR — top rear right
- SND_CHMAP_TRC — top rear center
- SND_CHMAP_TFLC — top front left center
- SND_CHMAP_TFRC — top front right center
- SND_CHMAP_TSL — top side left
- SND_CHMAP_TSR — top side right
- SND_CHMAP_LLFE — left LFE
- SND_CHMAP_RLFE — right LFE
- SND_CHMAP_BC — bottom center
- SND_CHMAP_BLC — bottom left center
- SND_CHMAP_BRC — bottom right center

Classification:

QNX Neutrino

Related Links

snd_pcm_get_chmap() (p. 323)

Get the current channel mapping, accounting for voice conversion on the capture path

snd_pcm_query_channel_map() (p. 410)

Get the current channel mapping for a PCM stream

snd_pcm_set_chmap() (p. 425)

Set the current channel mapping for a PCM stream

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINTR

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

-EINVAL

The state of *handle* is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_open() (p. 347)
: *Create a handle and open a connection to a specified audio interface*

snd_pcm_open_preferred() (p. 353)
: *Create a handle and open a connection to the preferred audio interface*

snd_pcm_event_t

Information about the PCM event that occurred

Synopsis:

```
typedef struct snd_pcm_event
{
    int32_t                          type;
    uint8_t                          zero[4];
    union
    {
        snd_pcm_status_event_t      audiomgmt_status;
        snd_pcm_mute_event_t        audiomgmt_mute;
        snd_pcm_outputclass_event_t outputclass;
        uint8_t                     reserved[128];
 }   data;
 uint8_t                         reserved[128];
} snd_pcm_event_t;
```

Description:

The `snd_pcm_event_t` structure describes information about event for a channel. You can use this structure to get information about the event. To retrieve and event, you call *snd_pcm_channel_read_event()*.

The members include:

type

One of the following types of event:

- SND_PCM_EVENT_AUDIOMGMT_STATUS — the status changed. To get more information about the event, you look at *data*, which is an *`snd_pcm_status_event_t`*.
- SND_PCM_EVENT_AUDIO_MGMT_MUTE — the channel has been muted because audio concurrency management has ducked the volumeto zero. To get more information about the event, you look at *data*, which is an *`snd_pcm_mute_event_t`*.
- SND_PCM_EVENT_OUTPUTCLASS_CHANGE — the output class has been changed. You may need to reconfigure the audio stream. To get more information about the event, you look at *data*, which is an *`snd_pcm_outputclass_event_t`*.
- SND_PCM_EVENT_OVERRUN — the capture channel is in an overrun state. For information on repreparing the channel, see *`snd_pcm_channel_status_t`*.
- SND_PCM_EVENT_UNDERRUN — the playback channel is in an underrun state. For information on repreparing the channel, see *`snd_pcm_channel_status_t`*.

data

The event, which is one of the following structure types:

- `snd_pcm_status_event_t` — a state change occurred.
- `snd_pcm_mute_event_t` — the channel was muted.
- `snd_pcm_outputclass_event_t` — the output class for the channel changed.

Classification:

QNX Neutrino

Related Links

snd_pcm_channel_read_event() (p. 274)
Get a PCM event

`snd_pcm_mute_event_t` (p. 344)
Information about the mute event

`snd_pcm_outputclass_event_t` (p. 356)
Information about the output port.

`snd_pcm_status_event_t` (p. 431)
Information about status change because audio concurrency management occurred on the audio stream

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

Library:

libasound.so

Use the `-l asound` option with `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. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

in the QNX Neutrino *C Library Reference*

snd_pcm_filter_t

The PCM Events to filter

Synopsis:

```
typedef struct snd_pcm_filter
{
   uint32_t  enable;
   uint8_t  reserved[124]; /* must be filled with zero */
} snd_pcm_filter_t;
```

Description:

The `snd_pcm_filter_t` structure is a filter that specifies which PCM events are filtered based on a bitmask. For more information about the bitmask, see " *PCM events*" in the Audio Architecture chapter of this guide. The members include:

enable

A bit field that specifies a mask of which PCM events types to enable. To enable the PCM event type, use these bits:

- Bit 0 — SND_PCM_EVENT_AUDIOMGMT_STATUS
- Bit 1 — SND_PCM_EVENT_AUDIOMGMT_MUTE
- Bit 2 — SND_PCM_EVENT_OUTPUTCLASS

For example, to enable SND_PCM_EVENT_AUDIOMGMT_MUTE events, use:

```
snd_pcm_filter_t filter.enable = 1 <<SND_PCM_EVENT_AUDIOMGMT_MUTE;
```

Classification:

QNX Neutrino

Related Links

snd_pcm_get_filter() (p. 325)
Retrieves the PCM filters that your application is subscribed to.

snd_pcm_set_filter() (p. 427)
Enable or disable the generation of PCM events

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.
- SND_PCM_FMT_U32_LE — unsigned 32-bit PCM little endian.
- SND_PCM_FMT_U32_BE — unsigned 32-bit PCM big endian.
- SND_PCM_FMT_S32_LE — signed 32-bit PCM little endian.
- SND_PCM_FMT_S32_BE — signed 32-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 with `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()*.

Library:

libasound.so

Use the `-l asound` option with `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

Safety:

Interrupt handler	No
Signal handler	Yes
Thread	Yes

Related Links

snd_pcm_build_linear_format() (p. 238)
Encode a linear format value

snd_pcm_format_linear() (p. 304)
Check for a linear format

snd_pcm_format_little_endian() (p. 306)
Check for a little-endian format

snd_pcm_format_signed() (p. 308)
Check for a signed format

snd_pcm_format_size() (p. 310)
Convert the size in the given samples to bytes

snd_pcm_format_unsigned() (p. 314)
Check for an unsigned format

snd_pcm_format_width() (p. 316)
Return the sample width in bits for a format

snd_pcm_get_format_name() (p. 327)
Convert a format value into a human-readable text string

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()*.

Library:

libasound.so

Use the `-l asound` option with `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_S24_4_LE
- SND_PCM_SFMT_U24_4_LE
- SND_PCM_SFMT_S24_4_BE
- SND_PCM_SFMT_U24_4_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()*.

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

Related Links

snd_pcm_build_linear_format() (p. 238)
Encode a linear format value

snd_pcm_format_big_endian() (p. 302)
Check for a big-endian format

snd_pcm_format_little_endian() (p. 306)
Check for a little-endian format

snd_pcm_format_signed() (p. 308)
Check for a signed format

snd_pcm_format_size() (p. 310)
Convert the size in the given samples to bytes

snd_pcm_format_unsigned() (p. 314)
Check for an unsigned format

snd_pcm_format_width() (p. 316)
Return the sample width in bits for a format

snd_pcm_get_format_name() (p. 327)
Convert a format value into a human-readable text string

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()*.

Library:

libasound.so

Use the `-l asound` option with `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

Safety:

Interrupt handler	No
Signal handler	Yes
Thread	Yes

Related Links

snd_pcm_build_linear_format() (p. 238)
  Encode a linear format value

snd_pcm_format_big_endian() (p. 302)
  Check for a big-endian format

snd_pcm_format_signed() (p. 308)
  Check for a signed format

snd_pcm_format_size() (p. 310)
  Convert the size in the given samples to bytes

snd_pcm_format_unsigned() (p. 314)
  Check for an unsigned format

snd_pcm_format_width() (p. 316)
  Return the sample width in bits for a format

snd_pcm_get_format_name() (p. 327)
  Convert a format value into a human-readable text string

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()*.

Library:

libasound.so

Use the `-l asound` option with `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

Safety:

Interrupt handler	No
Signal handler	Yes
Thread	Yes

Related Links

snd_pcm_build_linear_format() (p. 238)
Encode a linear format value

snd_pcm_format_big_endian() (p. 302)
Check for a big-endian format

snd_pcm_format_little_endian() (p. 306)
Check for a little-endian format

snd_pcm_format_size() (p. 310)
Convert the size in the given samples to bytes

snd_pcm_format_unsigned() (p. 314)
Check for an unsigned format

snd_pcm_format_width() (p. 316)
Return the sample width in bits for a format

snd_pcm_get_format_name() (p. 327)
Convert a format value into a human-readable text string

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()*.

num_samples

The number of samples for which you want to determine the size.

Library:

libasound.so

Use the `-l asound` option with `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

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Related Links

snd_pcm_build_linear_format() (p. 238)
: *Encode a linear format value*

snd_pcm_format_big_endian() (p. 302)
: *Check for a big-endian format*

snd_pcm_format_little_endian() (p. 306)
: *Check for a little-endian format*

snd_pcm_format_signed() (p. 308)
: *Check for a signed format*

snd_pcm_format_unsigned() (p. 314)
: *Check for an unsigned format*

snd_pcm_format_width() (p. 316)
: *Return the sample width in bits for a format*

snd_pcm_get_format_name() (p. 327)
: *Convert a format value into a human-readable text string*

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()*.

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` 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

Related Links

`snd_pcm_channel_info_t` (p. 259)
Information structure for a PCM channel

`snd_pcm_channel_params_t` (p. 266)
PCM channel parameters

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()*.

Library:

libasound.so

Use the `-l asound` option with `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

Safety:

Interrupt handler	No
Signal handler	Yes
Thread	Yes

Related Links

snd_pcm_build_linear_format() (p. 238)
    Encode a linear format value

snd_pcm_format_big_endian() (p. 302)
    Check for a big-endian format

snd_pcm_format_little_endian() (p. 306)
    Check for a little-endian format

snd_pcm_format_signed() (p. 308)
    Check for a signed format

snd_pcm_format_size() (p. 310)
    Convert the size in the given samples to bytes

snd_pcm_format_width() (p. 316)
    Return the sample width in bits for a format

snd_pcm_get_format_name() (p. 327)
    Convert a format value into a human-readable text string

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()*.

Library:

libasound.so

Use the `-l asound` option with `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
Thread	Yes

Related Links

snd_pcm_build_linear_format() (p. 238)
  Encode a linear format value

snd_pcm_format_big_endian() (p. 302)
  Check for a big-endian format

snd_pcm_format_little_endian() (p. 306)
  Check for a little-endian format

snd_pcm_format_signed() (p. 308)
  Check for a signed format

snd_pcm_format_size() (p. 310)
  Convert the size in the given samples to bytes

snd_pcm_format_unsigned() (p. 314)
  Check for an unsigned format

snd_pcm_get_format_name() (p. 327)
  Convert a format value into a human-readable text string

snd_pcm_free_chmaps()

Free a list of channel mappings

Synopsis:

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

void snd_pcm_free_chmaps( snd_pcm_chmap_query_t **maps );
```

Arguments:

maps

A pointer to the list of maps (of type `snd_pcm_chmap_query_t`) that you want to free.

Library:

libasound.so

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

Description:

The *snd_pcm_free_chmaps()* function frees a list of channel mappings that was returned by *snd_pcm_query_chmaps()*.

Examples:

See *snd_pcm_query_chmaps()*.

Classification:

QNX Neutrino

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

Related Links

`snd_pcm_chmap_query_t` (p. 290)
  Entry in an array of channel maps

snd_pcm_get_chmap() (p. 323)
  Get the current channel mapping, accounting for voice conversion on the capture path

snd_pcm_query_channel_map() (p. 410)
: *Get the current channel mapping for a PCM stream*

snd_pcm_query_chmaps() (p. 413)
: *Get a list of the available channel mappings for a PCM stream*

snd_pcm_set_chmap() (p. 425)
: *Set the current channel mapping for a PCM stream*

snd_pcm_get_apx_data()

Retrieve data from the acoustic library in an APX module

Synopsis:

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

int snd_pcm_set_apx_data( snd_pcm_t *pcm,
                          uint32_t apx_id,
                          snd_pcm_apx_param_t *param,
                          void *data );
```

Arguments:

pcm

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

apx_id

The ID of the APX module you want to access.

param

The library parameter information and the size of the data to set.

data

A pointer to the location to store the retrieved data. If *data* is NULL, the size member of *param* is still updated with the minimum size required for the data buffer.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_pcm_get_apx_data()* function retrieves data from the acoustic library in an APX, as specified in the `snd_pcm_apx_param_t` structure that *param* points to:

```
typedef struct snd_pcm_apx_param {
    uint32_t    size;
    uint32_t    dataId;
    int32_t      channel;
    int32_t      status;
} snd_pcm_apx_param_t;
```

This structure includes the following members:

size

The size of the buffer that the data argument points to for *snd_pcm_set_apx_data()* and *snd_pcm_get_apx_data()*. For *snd_pcm_get_apx_data()*, the *data* argument can be NULL. On return, the *size* member is updated with the actual size of the retrieved data.

dataId

The identifier for the parameter whose value you want to access.

channel

The channel identifier. Used only for selected channel-based parameters. The channel number is zero-based and should be from 0 to one less than the maximum number of channels valid for the specified parameter.

status

The return code from the library; one of the QWA_* codes defined in **qwa_err.h**.

Because this function is only used with acoustic (SFO, SPM) APX modules, it must be called on the playback PCM device.

Returns:

0 on success, or a negative *errno* value if an error occurred.

You should check both the return value of *snd_pcm_get_apx_data()* and the *status* member of `snd_pcm_apx_param_t` to determine if a call is successful.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

EINVAL

The handle *pcm* is not opened, the value of *param* is NULL, or *apx_id* is invalid.

ENOMEM

Not enough memory was available to copy the buffer.

ENODEV

The APX module specified by *apx_id* is not configured.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_load_apx_dataset() (p. 341)
Load an acoustic processing dataset into a running APX module

snd_pcm_set_apx_data() (p. 418)
Set data within the acoustic library in an APX module

snd_pcm_set_apx_external_volume() (p. 421)
Load volume controls external to `io-audio`

snd_pcm_set_apx_user_volume() (p. 423)
Load the user-intended volume

snd_pcm_get_chmap()

Get the current channel mapping, accounting for voice conversion on the capture path

Synopsis:

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

snd_pcm_chmap_t * snd_pcm_get_chmap( snd_pcm_t *pcm );
```

Arguments:

pcm

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

Library:

libasound.so

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

Description:

The *snd_pcm_get_chmap()* function gets the current channel mapping for a PCM stream, accounting for voice conversion on the capture path.

Returns:

A pointer to a `snd_pcm_chmap_t` structure that describes the mapping, or NULL if there isn't a mapping or an error occurred (*errno* is set). You must free this structure when you're done with it.

Errors:

EINVAL

The value of *pcm* is NULL.

ENOMEM

Not enough memory was available.

Examples:

```
snd_pcm_chmap_t *chmap;
int i;

printf("Get channel map using snd_pcm_get_chmap()...\n");
if ((chmap = snd_pcm_get_chmap(pcm_handle)) == NULL )
{
    fprintf(stderr, "snd_pcm_get_chmap failed: %s\n", snd_strerror(errno));
```

```
        return -1;
    }

    printf("Number of channels = %d\n", chmap->channels);
    for(i = 0; i < chmap->channels; i++)
        printf("\tchmap.pos[%d] = %d\n", i, chmap->pos[i]);

    free (chmap);
```

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function isn't thread-safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

`snd_pcm_chmap_t` (p. 291)
  Information about a channel map

snd_pcm_free_chmaps() (p. 318)
  Free a list of channel mappings

snd_pcm_query_channel_map() (p. 410)
  Get the current channel mapping for a PCM stream

snd_pcm_query_chmaps() (p. 413)
  Get a list of the available channel mappings for a PCM stream

snd_pcm_set_chmap() (p. 425)
  Set the current channel mapping for a PCM stream

snd_pcm_get_filter()

Retrieves the PCM filters that your application is subscribed to.

Synopsis:

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

int snd_pcm_get_filter( (snd_pcm_t *pcm,
                         int channel,
                         snd_pcm_filter_t * filter);
```

Arguments:

pcm

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

channel

The channel for the PCM handle. SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

filter

A pointer to a *snd_pcm_filter_t* structure that *snd_pcm_get_filter()* fills in with the mask.

Library:

libasound.so

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

Description:

The *snd_pcm_get_filter()* function fills the `snd_pcm_filter_t` structure with a mask of all PCM events for the channel that the handle was opened on.

You can arrange to have your application receive notification when an event occurs by calling *select()* on the channel, You can use *snd_pcm_channel_read_event()* to read the event's data.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

Invalid *pcm* or *filter* is NULL.

Classification:

QNX Neutrino

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

Related Links

`snd_pcm_filter_t` (p. 299)
The PCM Events to filter

snd_pcm_channel_read_event() (p. 274)
Get a PCM event

snd_pcm_set_filter() (p. 427)
Enable or disable the generation of PCM events

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 with `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; 3 bytes

SND_PCM_SFMT_U24_BE

Unsigned 24-bit Big Endian; 3 bytes

SND_PCM_SFMT_S24_LE

Signed 24-bit Little Endian; 3 bytes

SND_PCM_SFMT_S24_BE

Signed 24-bit Big Endian; 3 bytes

SND_PCM_SFMT_U24_4_LE

Unsigned 24-bit Little Endian; 4 bytes, right-aligned

SND_PCM_SFMT_U24_4_BE

Unsigned 24-bit Big Endian; 4 bytes, right-aligned

SND_PCM_SFMT_S24_4_LE

Signed 24-bit Little Endian; 4 bytes, right-aligned

SND_PCM_SFMT_S24_4_BE

Signed 24-bit Big Endian; 4 bytes, right-aligned

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

Related Links

snd_pcm_build_linear_format() (p. 238)
Encode a linear format value

snd_pcm_format_big_endian() (p. 302)
Check for a big-endian format

snd_pcm_format_little_endian() (p. 306)
: *Check for a little-endian format*

snd_pcm_format_signed() (p. 308)
: *Check for a signed format*

snd_pcm_format_size() (p. 310)
: *Convert the size in the given samples to bytes*

snd_pcm_format_unsigned() (p. 314)
: *Check for an unsigned format*

snd_pcm_format_width() (p. 316)
: *Return the sample width in bits for a format*

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

info

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

Library:

libasound.so

Use the `-l asound` option with `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

The state of *handle* is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

`snd_pcm_info_t` (p. 333)
Capability information about a PCM device

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

Related Links

snd_ctl_pcm_info() (p. 168)
Get general information about a PCM device from a control handle

snd_pcm_info() (p. 331)
Get general information about a PCM device

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

- *pcm1*—A PCM handle that may or may not belong to a group.
- *pcm2*—A PCM handle that must not belong to an existing group.

Library:

libasound.so

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

Description:

When you link two PCM streams using *snd_pcm_link()*, they always stream at the same time. Starting one stream causes the other streams in the group to start. You can call *snd_pcm_*_resume()*, *snd_pcm_*_pause(), and snd_pcm_*_prepare()* on the grouped PCM handles to start the audio stream. This default behavior is referred to as synchronized behavior and is the default PCM link mode (SND_PCM_LINK_MODE_SYNC). You can change this behavior using the *snd_pcm_link_mode()* and *snd_pcm_transition()* functions.

Regardless of the mode you choose, audio concurrency management policies are applied to the grouped PCM streams as single audio stream.

PCM handles for playback must have the same audio type for this function to create a new link to a group; otherwise the call fails. In addition, *pcm2* cannot already belong to a PCM link group. PCM handles for capture aren't required to be the same audio type.

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	Read the Caveats

Caveats:

This function is not thread safe if *pcm1* or *pcm2*(`snd_pcm_t`) are used across multiple threads.

Related Links

snd_pcm_link_transition() (p. 339)
: *Indicate to the specified PCM handle to trigger other handles in the group to start playback as soon as the specified audio subchannel's currently buffered data has been completely written to the hardware.*

snd_pcm_link_mode() (p. 337)
: *Set the link mode for a subchannel in a group*

snd_pcm_open() (p. 347)
: *Create a handle and open a connection to a specified audio interface*

snd_pcm_open_preferred() (p. 353)
: *Create a handle and open a connection to the preferred audio interface*

snd_pcm_unlink() (p. 434)
: *Detach a PCM stream from a link group*

snd_pcm_link_mode()

Set the link mode for a subchannel in a group

Synopsis:

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

int snd_pcm_link_mode(snd_pcm_t *handle, int mode);
```

Arguments:

handle

A handle to the subchannel in the PCM group.

mode

he behavior of how calls to one thread impact the other subchannels in the same group. You can use one of the following modes:

- SND_PCM_LINK_MODE_SYNC—(Default) The subchannels in the group synchronize together to respond to a call made to a subchannel in the group. For example, if you call *snd_pcm_*_resume()* on one subchannel in the group, the other subchannels in the group respond to the *snd_pcm_*_resume()* call and transition to the RUNNING state.
- SND_PCM_LINK_MODE_ASYNC—For audio concurrency management purposes, the subchannels that are grouped together are treated as a single audio stream, but synchronization of the calls made to a subchannel in the group won't occur. For example, if you call *snd_pcm_*_resume()*, only the specified subchannel resumes playing.
- SND_PCM_LINK_MODE_TRANSITION—Start playback on the PCM handles in the group that have this mode set when the current PCM handle's buffer is empty.

Library:

libasound.so

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

Description:

The *snd_pcm_link_mode()* function sets the link mode for subchannel in a group. Within the group, you can configure different subchannels to have different modes —in other words, you can mix the modes that are set on different PCM handles belonging to the same group. For example, if you have a group of four subchannels, you could set the mode of three of them to SND_PCM_LINK_MODE_SYNC and the remaining subchannel to SND_PCM_LINK_MODE_ASYNC.

Returns:

EOK on success, or an *errno* value if an error occurred.

Errors:

-EINVAL

Failed to change the link mode because of either an invalid handle to a PCM group or the application tried to set the link mode for a subchannel that's used to capture audio.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* is used across multiple threads.

snd_pcm_link_transition ()

Indicate to the specified PCM handle to trigger other handles in the group to start playback as soon as the specified audio subchannel's currently buffered data has been completely written to the hardware.

Synopsis:

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

int snd_pcm_link_transition (snd_pcm_t handle);
```

Arguments:

handle

A PCM handle that may or may not belong to a group.

Library:

libasound.so

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

Description:

Indicate to the specified PCM handle to trigger other handles in the group to start playback as soon as the specified audio subchannel's currently buffered data has been completely written to the hardware. This allows you to transition playback from one PCM handle to another with minimal gap between playback. For example, if you had two subchannels in a group and you set the mode to SND_PCM_LINK_MODE_TRANSITION for both, then when you call *snd_pcm_link_transition()* on the active PCM handle, as it nears completion, the other subchannel is triggered to start playing.

Returns:

EOK on success, or an *errno* value if an error occurred.

Errors:

-EINVAL

Failed to change the link mode because of either an invalid handle to a PCM group or the application tried to set the link mode for a subchannel that's used to capture audio.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* is used across multiple threads.

Related Links

snd_pcm_link_mode() (p. 337)
Set the link mode for a subchannel in a group

snd_pcm_link() (p. 335)
Link two PCM streams together

snd_pcm_load_apx_dataset()

Load an acoustic processing dataset into a running APX module

Synopsis:

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

int snd_pcm_load_apx_dataset( snd_pcm_t *pcm,
                              uint32_t apx_id,
                              const char *dataset,
                              int *ap_status );
```

Arguments:

pcm

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

apx_id

The ID of the APX module you want to access.

dataset

The name of the dataset to load.

ap_status

A pointer to the location to store the acoustic library return code.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_pcm_load_apx_dataset()* function loads an acoustic processing data set into a running APX. A **.conf** key *apx*`_dataset_qcf_`*dataset-name* is created to look up the acoustic data set filepath, where *apx* is the type of APX module and *dataset-name* is the string specified by the *dataset* argument. For example, for SFO and a dataset name `foo`: `sfo_dataset_qcf_foo`.

An error is returned if the data set file can't be found; otherwise, the name of the data set is stored and one of the following actions is taken:

- If the APX module is running, the data set is loaded immediately. The library return code is returned in the *ap_status* argument. An application should check both the return value of *snd_pcm_load_apx_dataset()* and the *ap_status* argument to determine if a call is successful.
- If the APX module isn't running, the data set is loaded when the APX starts, after the library is initialized with the qcf configuration file for the specified mode, but before processing starts. If an error occurs when loading the data set, the APX continues to run.

After it is set, the data set is loaded each time the APX module is started until a new data set is applied.

Because this function is only used with acoustic (SFO, SPM) APX modules, it must be called on the playback PCM device.

Returns:

EOK on success, or a negative *errno* value if an error occurred.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

EINVAL

One of the following causes:

- The handle *pcm* is not opened.
- The value of *apx_id* is not valid.
- The value of *pcm*, *dataset*, or *ap_status* is NULL.

ENOMEM

Not enough memory to copy the data set string was available.

ENODEV

The APX module specified by *apx_id* is not configured.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_get_apx_data() (p. 320)
Retrieve data from the acoustic library in an APX module

snd_pcm_set_apx_data() (p. 418)
Set data within the acoustic library in an APX module

snd_pcm_set_apx_external_volume() (p. 421)
Load volume controls external to `io-audio`

snd_pcm_set_apx_user_volume() (p. 423)
Load the user-intended volume

snd_pcm_mute_event_t

Information about the mute event

Synopsis:

```
typedef struct snd_pcm_mute_event
{
 uint8_t  mute;
 uint8_t  reason;
#define SND_PCM_MUTE_EVENT_SAME_PRIORITY     1
#define SND_PCM_MUTE_EVENT_HIGHER_PRIORITY   2
 uint8_t  zero[2];
} snd_pcm_mute_event_t;
```

Description:

The `snd_pcm_mute_event` structure is delivered whenever a channel is ducked to or unducked from zero and preemption isn't enabled. You can use this structure to get information to determine if you were ducked because a higher or same priority audio stream played.

The members include:

mute

The event indicates whether the channel was muted. A value of `1` specifies that the channel was muted; otherwise the channel was unmuted.

reason

A value of SND_PCM_MUTE_EVENT_SAME_PRIORITY specifies the audio type that caused the channel to mute is the same priority. A value of SND_PCM_MUTE_EVENT_HIGHER_PRIORITY specifies that the channel was ducked because the audio type was a higher priority.

Classification:

QNX Neutrino

Related Links

snd_pcm_channel_read_event() (p. 274)
  Get a PCM event

`snd_pcm_event_t` (p. 295)
  Information about the PCM event that occurred

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

nonblock

If this argument is nonzero, non-blocking mode is in effect for subsequent calls to *snd_pcm_read()* and *snd_pcm_write()*.

Library:

libasound.so

Use the `-l asound` option with `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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

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.

Related Links

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

snd_pcm_read() (p. 415)
Transfer PCM data from the capture channel

snd_pcm_write() (p. 438)
Transfer PCM data to playback channel

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()*

Library:

libasound.so

Use the `-l asound` option with `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()*) is preferred to using numbers (*snd_pcm_open()*), although *snd_pcm_open_preferred()*. 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*" in the Playing and Capturing Audio Data chapter.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

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()* or *snd_pcm_plugin_params()* .

Related Links

snd_pcm_close() (p. 293)
Close a PCM handle and free its resources

snd_pcm_nonblock_mode() (p. 345)
Set or reset the blocking behavior of reads and writes to PCM channels

snd_pcm_open_name() (p. 350)
: *Create a handle and open a connection to an audio interface specified by name*

snd_pcm_open_preferred() (p. 353)
: *Create a handle and open a connection to the preferred audio interface*

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. Can be one of the following names (all found under **/dev/snd**):

- a symbolic name (for example, `pcmPreferredp`, `pcmPreferredc`, or a name an *io-audio* option specifies).
- a system-assigned name that identifies the card and device (for example, `pcmC0D0c` or `pcmC0D0p`; see *Cards and devices* in the “Audio Architecture” chapter).

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()*.

Library:

libasound.so

Use the `-l asound` option with `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()*), although *snd_pcm_open_preferred()*. remains a good alternative to both.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *handle* is invalid, the mode is invalid, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was 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* " in the Playing and Capturing Audio Data chapter.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

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()* or *snd_pcm_plugin_params()* .

Related Links

snd_pcm_close() (p. 293)
Close a PCM handle and free its resources

snd_pcm_nonblock_mode() (p. 345)
Set or reset the blocking behavior of reads and writes to PCM channels

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

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()*.

Library:

libasound.so

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

Description:

The *snd_pcm_open_preferred()* function is an extension to the *snd_pcm_open()* function that attempts to open the user-selected default (or preferred) device for the system.

If you use this function, your application is more flexible than if you use *snd_pcm_open()*.

In a system where more than one PCM device exists, you can 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.

For information on how `io-audio` sets the preferred device, see "Preferred device selection" in the `io-audio` chapter of the *QNX Neutrino Utilities Reference*.

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*" in the Playing and Capturing Audio Data chapter.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

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()* or *snd_pcm_plugin_params()* .

Related Links

snd_pcm_close() (p. 293)
  Close a PCM handle and free its resources

snd_pcm_nonblock_mode() (p. 345)
  Set or reset the blocking behavior of reads and writes to PCM channels

snd_pcm_open() (p. 347)
  Create a handle and open a connection to a specified audio interface

snd_pcm_open_name() (p. 350)
  Create a handle and open a connection to an audio interface specified by name

snd_pcm_outputclass_event_t

Information about the output port.

Synopsis:

```
typedef struct snd_pcm_outputclass_event
{
    uint32_t  old_output_class;
    uint32_t  new_output_class;
} snd_pcm_outputclass_event_t;
```

Description:

The `snd_pcm_outputclass_event` structure describes information about the output class event. You can use this structure to get information about the outputclass for the PCM channel. You get handle about this structure from the *snd_pcm_event_t()* structure.

The members include:

old_output_class

The previous output class type. The valid values are:

- SND_OUTPUT_CLASS_UKNOWN — the output channel is for an unknown type.
- SND_OUTPUT_CLASS_SPEAKER — the output channel is for speakers that are connected to the system.
- SND_OUTPUT_CLASS_HEADPHONE — the output channel is for headphones that are connected to the system.
- SND_OUTPUT_CLASS_LINEOUT— the output channel is for the line-out channel of the system.
- SND_OUTPUT_CLASS_BLUETOOTH— the output channel is for Bluetooth.
- SND_OUTPUT_CLASS_TOSLINK — the output channel is for an S-Link connector.
- SND_OUTPUT_CLASS_MIRACAST — the output channel is for Miracast.
- SND_NUM_OUTPUT_CLASSES — an end-of-list identifier that indicates the total number of output types recognized by this library.

new_output_class

The current output class that the channel was changed to which caused the event to occur. The valid values are:

- SND_OUTPUT_CLASS_UKNOWN — the output channel is for an unknown type.
- SND_OUTPUT_CLASS_SPEAKER — the output channel is for speakers that are connected to the system.
- SND_OUTPUT_CLASS_HEADPHONE — the output channel is for headphones that are connected to the system.

- SND_OUTPUT_CLASS_LINEOUT — the output channel is for the line-out channel of the system.
- SND_OUTPUT_CLASS_BLUETOOTH — the output channel is for Bluetooth.
- SND_OUTPUT_CLASS_TOSLINK — the output channel is for an S-Link connector.
- SND_OUTPUT_CLASS_MIRACAST — the output channel is for Miracast.
- SND_NUM_OUTPUT_CLASSES — an end-of-list identifier that indicates the total number of output types recognized by this library.

Classification:

QNX Neutrino

Related Links

snd_pcm_channel_read_event() (p. 274)
Get a PCM event

`snd_pcm_event_t` (p. 295)
Information about the PCM event that occurred

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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, even if the channel's current state has been set to SND_PCM_STATUS_SUSPENDED or SND_PCM_STATUS_PAUSED by audio concurrency management policies in place.

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

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of the *handle* is invalid or an invalid state change occurred, or the PCM device state isn't ready. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_channel_flush() (p. 252)
    Flush all pending data in a PCM channel's queue and stop the channel

snd_pcm_playback_flush() (p. 360)
    Play out all pending data in a PCM playback channel's queue and stop the channel

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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.

Since this call is a blocking call, if the channel is in the SND_PCM_STATUS_SUSPENDED state an error occurs if you call this function. If you call this function while in the SND_PCM_STATUS_PAUSED state, playback is resumed and the channel goes into the SND_PCM_STATUS_RUNNING state immediately,and then ends in the SND_PCM_STATUS_READY state.

If the channel doesn't start playing (not enough data has been written to start playback), then this call silences the remainder of the buffer to satisfy the playback start requirements. Once the playback requirements are satisfied, playback starts and allows the data to be played out. Once the buffer empties, this call returns successfully.

If this call completes successfully, the channel is left in the SND_PCM_STATUS_READY state.

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:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EAGAIN

The state was SND_PCM_STATUS_SUSPENDED while trying to call this function. This error is also returned if the state was SND_PCM_STATUS_PAUSED, but resuming the playback failed.

-EBADFD

The PCM device state isn't ready.

-EAGAIN

The PCM device state is in the SND_PCM_STATUS_SUSPENDED or SND_PCM_STATUS_PAUSED state.

-EINTR

The operation was interrupted because of a system signal (such as a timer) or an error was returned asynchronously.

-EINVAL

The state of *handle* is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_flush() (p. 240)
: *Discard all pending data in a PCM capture channel's queue and stop the channel*

snd_pcm_channel_flush() (p. 252)
: *Flush all pending data in a PCM channel's queue and stop the channel*

snd_pcm_playback_drain() (p. 358)
: *Stop the PCM playback channel and discard the contents of its queue*

snd_pcm_plugin_flush() (p. 371)
: *Finish processing all pending data in a PCM channel's queue and stop the channel*

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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()*). 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()*.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-EAGAIN

Indicates that there's insufficient data in the buffer to play. You should write more data to the buffer.

-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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_go() (p. 242)
Start a PCM capture channel running

snd_pcm_channel_go() (p. 255)
Start a PCM channel running

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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 on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EINVAL

The state of handle is invalid or an invalid state change occurred. The other reason this value is returned is because the specified channel isn't being played.

-EIO

The channel isn't valid that was passed in was not set to SND_PCM_CHANNEL_PLAYBACK or SND_PCM_CHANNEL_CAPTURE.

-ENOTSUP

Pause isn't supported on the PCM device that's referenced by the PCM handle (*pcm*).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_open() (p. 347)
  Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
  Create a handle and open a connection to the preferred audio interface

snd_pcm_playback_resume() (p. 369)
  Resume a channel that was paused while playing back

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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()*) 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:

EOK on success, a negative error code, or a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-EBUSY

The channel is already running.

-EIO

The channel was not set to SND_PCM_CHANNEL_PLAYBACK or SND_PCM_CHANNEL_CAPTURE.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_prepare() (p. 246)
 Signal the driver to ready the capture channel

snd_pcm_channel_go() (p. 255)
 Start a PCM channel running

snd_pcm_channel_prepare() (p. 272)
 Signal the driver to ready the specified channel

snd_pcm_plugin_prepare() (p. 383)
 Signal the driver to ready the specified channel (plugin-aware)

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

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

Description:

The *snd_pcm_playback_resume()* function resumes a channel that was paused while playing back.

If the channel is in the SUSPENDED (SND_PCM_STATUS_SUSPENDED), one of the following occurs when you call this function:

- If the channel is hard suspended, calling this function clears any underlying paused condition (e.g., *snd_pcm_channel_pause()* was called before moving to the SND_PCM_STATUS_SUSPENDED state). However, because you are in the SND_PCM_STATUS_SUSPENDED state and waiting for higher priority audio stream to complete, the channel remains in the SUSPENDED state.
- If the channel is in the soft suspended state, calling this function clears any underlying paused condition (e.g., *snd_pcm_channel_pause()* was called before moving to SND_PCM_STATUS_SUSPENDED state) and the channel plays immediately and transitions to the SND_PCM_STATUS_RUNNING state.
- If both the hard suspended and soft suspended conditions occur, the hard suspended state takes precedence. In other words, calling the function clears any pending paused conditione.g., *snd_pcm_channel_pause()*) but the channel remains in the SND_PCM_STATUS_SUSPENDED state.

For more information, see the *Audio Concurrency Management* chapter in this guide.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

Additional information for common error values:

-EAGAIN

Indicates that there's insufficient data in the buffer to play. You should write more data to the buffer.

-EINVAL

The state of *handle* is invalid, the channel wasn't being used for playback, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-ENOTSUP

Resume isn't supported on the PCM device that's referenced by the PCM handle (*pcm*).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

snd_pcm_playback_pause() (p. 365)
Pause a channel that's playing back

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

Library:

libasound.so

Use the `-l asound` option with `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 you use this call to process playback data, there are two considerations:

- Since this call is a blocking call, if the channel is in the SND_PCM_STATUS_SUSPENDED state, an error occurs if you call this function.
- If this function is called while in the SND_PCM_STATUS_PAUSED state while processing playback data, playback is resumed and the channel goes into the SND_PCM_STATUS_RUNNING state immediately, and then ends in the SND_PCM_STATUS_READY state.

  If the channel doesn't start playing (not enough data has been written to start playback), then this call silences the remainder of the buffer to satisfy the playback start requirements. Once the playback requirements are satisfied, playback starts and allows the data to be played out. Once the buffer empties, this call returns successfully.

If this call completes successfully, the channel is left in the SND_PCM_STATUS_READY state.

Returns:

A positive number on success, or a negative value on error.

Errors:

-EAGAIN

The state was SND_PCM_STATUS_SUSPENDED while trying to call this function. This error is also returned if the state was SND_PCM_STATUS_PAUSED, but resuming the playback failed.

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input,or an invalid state change occurred. You can call *snd_pcm_plugin_status()* to check if the state change was invalid.

-EINTR

The operation was interrupted because of a system signal (such as a timer) or an error was returned asynchronously.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

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()* . 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.

Related Links

snd_pcm_capture_flush() (p. 240)
: *Discard all pending data in a PCM capture channel's queue and stop the channel*

snd_pcm_channel_flush() (p. 252)
: *Flush all pending data in a PCM channel's queue and stop the channel*

snd_pcm_playback_flush() (p. 360)
: *Play out all pending data in a PCM playback channel's queue and stop the channel*

snd_pcm_plugin_playback_drain() (p. 381)
: *Stop the PCM playback channel and discard the contents of its queue (plugin-aware)*

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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` structure that the function fills in.

Library:

libasound.so

Use the `-l asound` option with `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 on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, *voice_conversion* was invalid, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-ENOENT

The voice converter plugin doesn't exist.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_plugin_set_voice_conversion() (p. 399)
Set the current voice conversion structure for a channel

`snd_pcm_voice_conversion_t` (p. 436)
Data structure that controls voice conversion

"Controlling voice conversion" (p. 36)
Configuration of the **libasound** library is based on the maximum number of voices supported in hardware.

in the Playing and Capturing Audio Data chapter

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

info

A pointer to a `snd_pcm_channel_info_t` 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 with `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()*, 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()*, which gets information about the *complete* capabilities of the system.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *handle* is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

This function is the plugin-aware version of *snd_pcm_channel_info()* . 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.

Related Links

snd_pcm_channel_info() (p. 257)
Get information about a PCM channel's current capabilities

`snd_pcm_channel_info_t` (p. 259)
Information structure for a PCM channel

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

params

A pointer to a `snd_pcm_channel_params_t` 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 with `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()*.

When you call *snd_pcm_plugin_params()*, you must provide the fragment size you would like to use. The `io-audio` services attempts to fulfill the requested fragment size however, many factors cause `io-audio` to return a size that doesn't exactly match the size that you requested, such as hardware/software alignment restrictions, required data conversions, etc. To determine the actual fragment size that the subchannel is configured for, you must call *snd_pcm_plugin_setup()*.

A software alignment restriction includes interfacing to the PCM Software mixer or PCM Input Splitter which operates at a fixed fragment period. For more information, see "*PCM software mixer*" and "*PCM input splitter*."

Example calculations for requesting a fragment size that holds 16ms of data

In this example the source audio data is 16Khz, 16-bit, mono and the audio playback device is 48Khz, 16-bit stereo:

```
Fragment size (in bytes) = (16000Hz * 2 bytes * 1 channels * 16ms) / 1000
                         = 512000 / 1000
                         = 512
```

Because the data's format doesn't match the configuration of the playback device, the **libasound** library will automatically up convert the data for you. This *up conversion* will result in your original 512 bytes of data becoming 3072 bytes of data at the device level. You can always base your fragment size on a time period rather than a fixed number of bytes so that you don't have to worry about any required data conversions changing the intended playback timings. For example:

```
App fragment period        = 512 / (16000 * 2 * 1 / 1000) = 16ms
libasound up conversions   = 512 * (48000/16000 * 2/1) = 3072 bytes
Playback fragment period   = 3072 / (48000 * 2 * 2 / 1000) = 16ms
```

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *handle* is invalid, the format is unsupported, or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

This function is the plugin-aware version of *snd_pcm_channel_params()* . 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.

Related Links

snd_pcm_channel_params() (p. 264)
: *Set a PCM channel's configurable parameters*

`snd_pcm_channel_params_t` (p. 266)
: *PCM channel parameters*

snd_pcm_channel_setup() (p. 279)
: *Get the current configuration for the specified PCM channel*

snd_pcm_open() (p. 347)
: *Create a handle and open a connection to a specified audio interface*

snd_pcm_open_preferred() (p. 353)
: *Create a handle and open a connection to the preferred audio interface*

snd_pcm_plugin_setup() (p. 401)
: *Get the current configuration for the specified PCM channel (plugin aware)*

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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, even if the channel's current state has been set to SND_PCM_STATUS_SUSPENDED or SND_PCM_STATUS_PAUSED by audio concurrency management policies in place.

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

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, an invalid state change occurred, or the PCM device state isn't ready. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

This function is the plugin-aware version of *snd_pcm_playback_drain()* . 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.

Related Links

snd_pcm_playback_drain() (p. 358)
Stop the PCM playback channel and discard the contents of its queue

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel; SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

Library:

libasound.so

Use the `-l asound` option with `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()*) 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:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EBUSY

The subchannel is in the running state.

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, or an invalid state change occurred. You can call *snd_pcm_plugin_status()* to check if the state change was invalid.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

This function is the plugin-aware version of *snd_pcm_channel_prepare()* . 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.

Related Links

snd_pcm_capture_prepare() (p. 246)
Signal the driver to ready the capture channel

snd_pcm_channel_go() (p. 255)
Start a PCM channel running

snd_pcm_channel_prepare() (p. 272)
Signal the driver to ready the specified channel

snd_pcm_playback_prepare() (p. 367)
Signal the driver to ready the playback channel

snd_pcm_plugin_reset_voice_conversion()

Resets the parameters of the voice conversion

Synopsis:

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

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

Arguments:

pcm

The handle returned by a call to a *snd_pmc_open_** function, such as *snd_pcm_open_name()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

channel

The channel to reset.

Library:

libasound.so

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

Description:

The *snd_pcm_plugin_reset_voice_conversion*() resets all previously settings of voice conversions to default.

Returns:

EOK upon success, a negative `errno` upon failure. The `errno` values are available in the **errno.h** file.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function isn't thread safe if the *pcm* argument(`snd_pcm_t`)is used across multiple threads.

Related Links

snd_pcm_open() (p. 347)
: *Create a handle and open a connection to a specified audio interface*

snd_pcm_open_preferred() (p. 353)
: *Create a handle and open a connection to the preferred audio interface*

snd_pcm_open_name() (p. 350)
: *Create a handle and open a connection to an audio interface specified by name*

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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 with `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()*.

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()*) 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 *errno* if an error occurred. The *errno* values are available in the **errno.h** file.

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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

This function is the plugin-aware version of *snd_pcm_read()* . 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_BUFFER_PARTIAL_BLOCKS bit with *snd_pcm_plugin_set_disable()* , 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).

Related Links

snd_pcm_plugin_set_disable() (p. 390)
Disable PCM plugins

snd_pcm_plugin_write() (p. 407)
Transfer PCM data to playback channel (plugin-aware)

snd_pcm_read() (p. 415)
Transfer PCM data from the capture channel

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  plugins );
```

Arguments:

pcm

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

plugins

A bitmap of the plugins that you want to disable; a bitwise OR of zero or more of the following (which are the plugins that are currently enabled by default):

- PLUGIN_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_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()*, *snd_pcm_channel_setup()*, and *snd_pcm_channel_status()*. These now behave as *snd_pcm_plugin_params()*, *snd_pcm_plugin_setup()*, and *snd_pcm_plugin_status()*, unless you've disabled the conversion by calling:

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

- PLUGIN_MMAP — disable the mmap plugins.

If mmap plugins are used, some of the members of the `snd_pcm_channel_status_t` structure aren't used.

Library:

libasound.so

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

Description:

You can use *snd_pcm_plugin_set_disable()* to disable plugins that are ordinarily used in the plugin chain. You need to do this before calling *snd_pcm_plugin_params()*.

Returns:

The updated bitmap of disabled plugins.

This function doesn't return a negative error code the way that other *snd_pcm_*()* functions do.

Examples:

See the *wave.c example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function isn't thread safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

`snd_pcm_channel_status_t` (p. 286)
PCM channel status structure

snd_pcm_plugin_read() (p. 387)
Transfer PCM data from the capture channel (plugin-aware)

snd_pcm_plugin_set_enable() (p. 393)
: *Enable PCM plugins*

snd_pcm_plugin_write() (p. 407)
: *Transfer PCM data to playback channel (plugin-aware)*

snd_pcm_plugin_set_enable()

Enable PCM plugins

Synopsis:

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

unsigned int snd_pcm_plugin_set_enable( snd_pcm_t   *pcm,
                                        unsigned int plugins );
```

Arguments:

pcm

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

plugins

A bitmap of the plugins that you want to enable.

Library:

libasound.so

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

Description:

You can use *snd_pcm_plugin_set_enable()* to enable plugins that aren't ordinarily used in the plugin chain. You need to do this before calling *snd_pcm_plugin_params()*.

Returns:

The updated bitmap of disabled plugins.

This function doesn't return a negative error code the way that other *snd_pcm_*()* functions do.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No

Safety:	
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function isn't thread safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

snd_pcm_plugin_set_disable() (p. 390)
Disable PCM plugins

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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)
- 2 — 20-point band pass

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()*, so that the plugin can be properly initialized (including the filters).

Because the polyphase SRC plugin converter provides the best quality rate conversion with the least amount of CPU usage, QSA always tries to use it as the source filter method, regardless of the filter set using *snd_pcm_plugin_set_src_method()*. For more information, see *"Polyphase SRC plugin converter"*.

Returns:

The current method.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

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.

Related Links

snd_pcm_plugin_params() (p. 378)
Set the configurable parameters for a PCM channel (plugin-aware)

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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 with `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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

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.

Related Links

snd_pcm_plugin_update_src() (p. 405)
   Get the size of the next fragment to write (plugin-aware)

snd_pcm_plugin_src_max_frag()

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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` structure that specifies how to convert the voices.

Library:

libasound.so

Use the `-l asound` option with `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. In QNX Neutrino 6.6 or later, this function instantiates the voice converter plugin if it doesn't already exist.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *handle* is invalid, an invalid *channel* was provided as input, or an invalid state change occurred. You can call *snd_pcm_plugin_status()* to check if the state change was invalid.

-ENOENT

The voice converter plugin doesn't exist.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_plugin_get_voice_conversion() (p. 374)
Get the current voice conversion structure for a channel

`snd_pcm_voice_conversion_t` (p. 436)
Data structure that controls voice conversion

"Controlling voice conversion" (p. 36)
Configuration of the **libasound** library is based on the maximum number of voices supported in hardware.

in the Playing and Capturing Audio Data chapter

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

setup

A pointer to a `snd_pcm_channel_setup_t` 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 with `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:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

Invalid *handle*; data pointer is NULL; *setup->mode* isn't SND_PCM_MODE_BLOCK.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

This function is the plugin-aware version of *snd_pcm_channel_setup()* . 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.

Related Links

snd_pcm_channel_params() (p. 264)
Set a PCM channel's configurable parameters

snd_pcm_channel_setup() (p. 279)
Get the current configuration for the specified PCM channel

`snd_pcm_channel_setup_t` (p. 281)
Current configuration of a PCM channel

`snd_mixer_gid_t` (p. 202)
Mixer group ID structure

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

status

A pointer to a `snd_pcm_channel_status_t` 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 with `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:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EBADFD

The pcm device state isn't ready.

-EFAULT

Failed to copy data.

-EINVAL

Invalid *handle* or the data pointer is NULL.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

This function is the plugin-aware version of *snd_pcm_channel_status()* . 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.

Related Links

snd_pcm_channel_status() (p. 284)
Get the runtime status of a PCM channel

`snd_pcm_channel_status_t` (p. 286)
PCM channel status structure

snd_pcm_open() (p. 347)
Create a handle and open a connection to a specified audio interface

snd_pcm_open_preferred() (p. 353)
Create a handle and open a connection to the preferred audio interface

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

setup

A pointer to a `snd_pcm_channel_setup_t` 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 with `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()*.

If you're using SND_SRC_MODE_ACTUAL or SND_SRC_MODE_ASYNC mode (see *snd_pcm_plugin_set_src_mode()*), 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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

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.

Related Links

snd_pcm_plugin_src_max_frag()

snd_pcm_plugin_set_src_mode() (p. 397)
Set the system's source mode (plugin-aware)

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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 with `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()* 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()*), you need to call *snd_pcm_plugin_update_src()* 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()*.

Errors:

-EAGAIN

Try again later. The subchannel is opened nonblock. When this function was called, it was in the SND_PCM_STATUS_PAUSED or SND_PCM_STATUS_SUSPENDED and the buffer is full.

-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` structure and PCM subbuffering has been disabled with *snd_pcm_plugin_set_disable()*.

-EWOULDBLOCK

The write would have blocked (nonblocking write).

Examples:

See the *wave.c example* in the appendix.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

This function is the plugin-aware version of *snd_pcm_write()* . 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_BUFFER_PARTIAL_BLOCKS bit with *snd_pcm_plugin_set_disable()* , 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).

Related Links

snd_pcm_plugin_read() (p. 387)
Transfer PCM data from the capture channel (plugin-aware)

snd_pcm_plugin_set_disable() (p. 390)
Disable PCM plugins

snd_pcm_plugin_set_src_mode() (p. 397)
Set the system's source mode (plugin-aware)

snd_pcm_plugin_update_src() (p. 405)
Get the size of the next fragment to write (plugin-aware)

snd_pcm_write() (p. 438)
Transfer PCM data to playback channel

snd_pcm_query_channel_map()

Get the current channel mapping for a PCM stream

Synopsis:

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

int snd_pcm_query_channel_map( snd_pcm_t *pcm,
                               snd_pcm_chmap_t *chmap );
```

Arguments:

pcm

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

chmap

A pointer to a `snd_pcm_chmap_t` structure where the function can store the mapping. The map contains playback or capture channels. You must allocate this structure before calling this function.

Library:

libasound.so

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

Description:

This function retrieves a map of speakers or microphones based on whether it's a playback or capture channel. When this function successfully returns, the map array contains a map of speakers. If you don't specify an array that's large enough to store the number of channels in map.pos, the function returns with ENOMEM. When the function returns with ENOMEM, `map.channels` is updated with the minimum size of array that is required for `map.pos`.

If you want to determine the correct size for the array, you can call this function the first time, to determine the array size to allocate for `map.pos`. Then, call the function a second time to retrieve the PCM channels.

Returns:

EOK on success, an `errno` upon failure. The `errno` values are available in the **errno.h** file.

Errors:

-EINVAL

The value of *pcm* or *chmap* is NULL.

-ENOMEM

Not enough memory was available.

Examples:

```
snd_pcm_chmap_t *chmap;
int i;

/* Size the chmap to hold up to 32 channels */

printf("Get channel map using snd_pcm_query_channel_map()...\n");
chmap = calloc(1, sizeof(snd_pcm_chmap_t) + (sizeof(int) * 32));
chmap->channels = 32;

if ((rtn = snd_pcm_query_channel_map(pcm_handle, chmap)) != EOK)
{
    fprintf(stderr, "snd_pcm_query_channel_map failed: %s\n", snd_strerror(rtn));
    return -1;
}

printf("Number of channels = %d\n", chmap->channels);
for(i = 0; i < chmap->channels; i++)
    printf("\tchmap.pos[%d] = %d\n", i, chmap->pos[i]);

free (chmap);
```

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function isn't thread-safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

`snd_pcm_chmap_t` (p. 291)
  Information about a channel map

snd_pcm_free_chmaps() (p. 318)
  Free a list of channel mappings

snd_pcm_get_chmap() (p. 323)

Get the current channel mapping, accounting for voice conversion on the capture path

snd_pcm_query_chmaps() (p. 413)

Get a list of the available channel mappings for a PCM stream

snd_pcm_set_chmap() (p. 425)

Set the current channel mapping for a PCM stream

snd_pcm_query_chmaps()

Get a list of the available channel mappings for a PCM stream

Synopsis:

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

snd_pcm_chmap_query_t **snd_pcm_query_chmaps( snd_pcm_t *pcm );
```

Arguments:

pcm

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

Library:

libasound.so

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

Description:

The *snd_pcm_query_chmaps()* function returns a list of the available channel mappings for a PCM stream. To free this list, call *snd_pcm_free_chmaps()*.

Returns:

A pointer to an array of `snd_pcm_chmap_query_t` structures that describe the mappings, or NULL if no mappings are available or an error occurred (*errno* is set).

Errors:

EINVAL

The value of *pcm* is NULL.

ENOMEM

Not enough memory was available.

Examples:

```
snd_pcm_chmap_query_t **chmaps;
snd_pcm_chmap_t *chmap;
int i, j;

/* Check available channel maps */
printf("Get all available channel maps...\n");
if ((chmaps = snd_pcm_query_chmaps(pcm_handle)) == NULL)
```

```
{
    fprintf(stderr, "snd_pcm_query_chmaps failed: %s\n", snd_strerror(rtn));
    return -1;
}

for (i=0; chmaps[i] != NULL; i++)
{
    printf("chmaps[%d]:\n", i);
    printf("Map Type = %d\n", chmaps[i]->type);
    printf("Number of channels = %d\n", chmaps[i]->map.channels);
    for(j = 0; j < chmaps[i]->map.channels; j++)
        printf("\tchmaps[%d].pos[%d] = %d\n", i, j, chmaps[i]->map.pos[j]);
}

snd_pcm_free_chmaps(chmaps);
```

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function isn't thread-safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_chmap_query_t (p. 290)
  Entry in an array of channel maps

snd_pcm_free_chmaps() (p. 318)
  Free a list of channel mappings

snd_pcm_get_chmap() (p. 323)
  Get the current channel mapping, accounting for voice conversion on the capture path

snd_pcm_query_channel_map() (p. 410)
  Get the current channel mapping for a PCM stream

snd_pcm_set_chmap() (p. 425)
  Set the current channel mapping for a PCM stream

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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 with `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()* or *snd_pcm_capture_prepare()*.

This function may suspend the client application if the blocking behavior is active (see *snd_pcm_nonblock_mode()*) 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` 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

The state of *handle* is invalid or an invalid state change occurred. You can call *snd_pcm_channel_status()* to check if the state change was invalid.

-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	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_capture_prepare() (p. 246)
Signal the driver to ready the capture channel

snd_pcm_channel_prepare() (p. 272)
Signal the driver to ready the specified channel

`snd_pcm_format_t` (p. 312)
PCM data format structure

snd_pcm_plugin_read() (p. 387)
Transfer PCM data from the capture channel (plugin-aware)

snd_pcm_write() (p. 438)

Transfer PCM data to playback channel

snd_pcm_set_apx_data()

Set data within the acoustic library in an APX module

Synopsis:

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

int snd_pcm_set_apx_data( snd_pcm_t *pcm,
                          uint32_t apx_id,
                          snd_pcm_apx_param_t *param,
                          const void *data );
```

Arguments:

pcm

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

apx_id

The ID of the APX module you want to access.

param

The library parameter information and the size of the data to set.

data

A pointer to the location of the data to set.

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_pcm_set_apx_data()* function retrieves data from the acoustic library in an APX, as specified in the `snd_pcm_apx_param_t` structure that *param* points to. For information on `snd_pcm_apx_param_t`, see *snd_pcm_get_apx_data()*.

This API should not be used to set the external or user volume in the library because the change will not be reflected in the structures `io-audio` uses to automatically calculate these values and the change will be overwritten. Instead, use *snd_pcm_set_apx_external_volume()* and *snd_pcm_set_apx_user_volume()*.

Because this function is only used with acoustic (SFO, SPM) APX modules, it must be called on the playback PCM device.

Returns:

0 on success, or a negative *errno* value if an error occurred.

You should check both the return value of *snd_pcm_get_apx_data()* and the *status* member of `snd_pcm_apx_param_t` to determine if a call is successful.

This function can also return the return values of *devctl()* (see *devctl()* in the QNX Neutrino *C Library Reference*).

Errors:

EINVAL

One of the following causes:

- The handle *pcm* is not opened.
- The value of *apx_id* is not valid.
- The value of *pcm* or *data* is NULL.

ENOMEM

Not enough memory was available to copy the buffer.

ENODEV

The APX module specified by *apx_id* is not configured.

Classification:

QNX Neutrino

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

Caveats:

This function is not thread safe if the handle (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_get_apx_data() (p. 320)
 Retrieve data from the acoustic library in an APX module

snd_pcm_load_apx_dataset() (p. 341)
 Load an acoustic processing dataset into a running APX module

snd_pcm_set_apx_external_volume() (p. 421)
 Load volume controls external to `io-audio`

snd_pcm_set_apx_user_volume() (p. 423)
 Load the user-intended volume

snd_pcm_set_apx_external_volume()

Load volume controls external to `io-audio`

Synopsis:

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

int snd_pcm_set_apx_external_volume( snd_pcm_t *pcm,
                                     int nVoices,
                                     int16_t *vol_mB );
```

Arguments:

pcm

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

nVoices

The number of voices that is inside the *vol_mB* buffer. Can be -1 to set all voices to one value. If *nVoices* is greater than 1, it is assumed that the volume parameter provided is an array of size *nVoices* (i.e., that specifies a separate volume for each channel).

vol_mB

A pointer to an array of volumes in millibels (mB).

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_pcm_set_apx_external_volume()* function allows clients to let an APX module know about volume controls that are external to `io-audio` (e.g., an external amplifier).

The *nVoices* and *vol_mB* arguments allow you to specify an array for cases where there are differences among channels. The external volume is used in SPM's speaker protection as well as SFO's default user volume calculation (when it is not overriden).

Because this function is only used with acoustic (SFO, SPM) APX modules, it must be called on the playback PCM device.

Returns:

EOK on success, or a negative *errno* value if an error occurred.

This function can also return the negative values that *ioctl()* can assign to *errno* (see *ioctl()* in the QNX Neutrino *C Library Reference*).

Errors:

EINVAL

One of the following causes:

- The handle *pcm* is not opened.
- The value of *pcm* or *vol_mB* is NULL.
- The software mixer is not set up on the *pcm* device.

ENODEV

No APX module is configured.

Classification:

QNX Neutrino

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

Related Links

snd_pcm_get_apx_data() (p. 320)
Retrieve data from the acoustic library in an APX module

snd_pcm_load_apx_dataset() (p. 341)
Load an acoustic processing dataset into a running APX module

snd_pcm_set_apx_data() (p. 418)
Set data within the acoustic library in an APX module

snd_pcm_set_apx_user_volume() (p. 423)
Load the user-intended volume

snd_pcm_set_apx_user_volume()

Load the user-intended volume

Synopsis:

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

int snd_pcm_set_apx_user_volume( snd_pcm_t *pcm,
                                 int16_t vol_mB );
```

Arguments:

pcm

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

vol_mB

The user volume in millibels (mB).

Library:

libasound.so

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

Description:

This function can only be used if you have QNX Acoustic Management Platform 3.0 installed.

The *snd_pcm_set_apx_user_volume()* function allows applications to provide an APX module with the volume set by the user. This user volume overrides SFO's default behavior, which is to calculate the volume across all common mixers and taking the maximum channel. After the user volume is set, SFO stops this default calculation and it is expected that changes to the user volume will use this call.

Because this function is only used with acoustic (SFO, SPM) APX modules, it must be called on the playback PCM device.

Returns:

EOK on success, or a negative *errno* value if an error occurred.

This function can also return the negative values that *ioctl()* can assign to *errno* (see *ioctl()* in the QNX Neutrino *C Library Reference*).

Errors:

EINVAL

One of the following causes:

- The handle *pcm* is not opened.
- The value of *pcm* is NULL.
- The software mixer is not set up on the *pcm* device.

ENODEV

SFO is not configured.

Classification:

QNX Neutrino

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

Related Links

snd_pcm_get_apx_data() (p. 320)
Retrieve data from the acoustic library in an APX module

snd_pcm_load_apx_dataset() (p. 341)
Load an acoustic processing dataset into a running APX module

snd_pcm_set_apx_data() (p. 418)
Set data within the acoustic library in an APX module

snd_pcm_set_apx_external_volume() (p. 421)
Load volume controls external to `io-audio`

snd_pcm_set_chmap()

Set the current channel mapping for a PCM stream

Synopsis:

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

int snd_pcm_set_chmap( snd_pcm_t *pcm,
                       const snd_pcm_chmap_t *map );
```

Arguments:

pcm

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

chmap

A pointer to a `snd_pcm_chmap_t` structure that specifies the new mapping.

Library:

libasound.so

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

Description:

The *snd_pcm_set_chmap()* function sets the current channel mapping for a PCM stream. It's supported only if the hardware is capable of being remapped.

Returns:

0 on success, or a negative *errno* value if an error occurred.

Errors:

EINVAL

One of the following:

- The value of *pcm* or *chmap* is NULL.
- The encoded number of voices is the mapping was greater than the maximum voices that the device supports.

ENOMEM

Not enough memory was available.

ENOTSUP

The device supports only one fixed mapping.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function isn't thread-safe if *pcm* (`snd_pcm_t`) is used across multiple threads.

Related Links

`snd_pcm_chmap_t` (p. 291)
Information about a channel map

snd_pcm_free_chmaps() (p. 318)
Free a list of channel mappings

snd_pcm_get_chmap() (p. 323)
Get the current channel mapping, accounting for voice conversion on the capture path

snd_pcm_query_channel_map() (p. 410)
Get the current channel mapping for a PCM stream

snd_pcm_query_chmaps() (p. 413)
Get a list of the available channel mappings for a PCM stream

snd_pcm_set_filter()

Enable or disable the generation of PCM events

Synopsis:

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

int snd_pcm_set_filter(snd_pcm_t *handle,
                       int channel,
                       const snd_pcm_filter_t * filter)
```

Arguments:

handle

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

channel

The channel to use. SND_PCM_CHANNEL_CAPTURE or SND_PCM_CHANNEL_PLAYBACK.

filter

A `const` pointer to a *`snd_pcm_filter_t`* structure that defines a mask of events to track.

Library:

libasound.so

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

Description:

The *snd_pcm_set_filter()* function uses the *`snd_pcm_filter_t`* structure to set the mask of all PCM events for the channel that the handle was opened on. 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 channel. You can use *snd_pcm_channel_read_event()* to read the event's data.

Returns:

EOK on success, a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

Invalid *handle* or *filter* is NULL.

Classification:

QNX Neutrino

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

Related Links

`snd_pcm_filter_t` (p. 299)
The PCM Events to filter

snd_pcm_channel_read_event() (p. 274)
Get a PCM event

snd_pcm_get_filter() (p. 325)
Retrieves the PCM filters that your application is subscribed to.

snd_pcm_set_output_class()

Set the current audio output port information.

Synopsis:

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

int snd_pcm_set_output_class( snd_pcm_t *pcm,
                              uint32_t outputclass);
```

Arguments:

pcm

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

outputclass

The output class to set the channel to, which can be one of the following values:

SND_OUTPUT_CLASS_UKNOWN

The output channel is for an unknown type.

SND_OUTPUT_CLASS_SPEAKER

Indicates that the output channel is for speakers that are connected to the system.

SND_OUTPUT_CLASS_HEADPHONE

Indicates that the output channel is for headphones that are connected to the system.

SND_OUTPUT_CLASS_LINEOUT

Indicates that the output channel is for the line-out channel of the system.

SND_OUTPUT_CLASS_BLUETOOTH

Indicates that the output channel is for Bluetooth.

SND_OUTPUT_CLASS_TOSLINK

Indicates the output channel is for an S-Link connector.

SND_OUTPUT_CLASS_MIRACAST

Indicates that the output channel is for Miracast.

SND_NUM_OUTPUT_CLASSES

An end-of-list identifier that indicates the total number of output types recognized by this library.

Library:

libasound.so

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

Description:

The *snd_pcm_set_output_class()* function sets the output class for the PCM playback device. This call is a privileged call. Applications must be root or be part of same user group as `io-audio` is running as. The call is used only if the audio output routing isn't controlled by the audio driver directly (e.g., an external amplifier). External applications which control the audio output routing should use this call to notify `io-audio` of the change.

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

Returns:

EOK on success, EINVAL for invalid argument values passed to this function, or a negative a negative *errno* upon failure. The *errno* values are available in the **errno.h** file.

Errors:

-EINVAL

The state of *pcm* isn't valid or an the playback channel isn't ready, or the specified output class couldn't be set.

-EPERM

An application doesn't have the appropriate privileges to call this function.

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

snd_pcm_status_event_t

Information about status change because audio concurrency management occurred on the audio stream

Synopsis:

```
typedef struct snd_pcm_status_event
{
 uint32_t old_status;
 uint32_t new_status;
 uint32_t flags;
#define SND_PCM_STATUS_EVENT_HARD_SUSPEND (1<<0)
#define SND_PCM_STATUS_EVENT_SOFT_SUSPEND (1<<1)
#define SND_PCM_STATUS_EVENT_AUTOPAUSE    (1<<2)
} snd_pcm_status_event_t;
```

Description:

The `snd_pcm_status_event` structure describes status information about the event. You can use this structure to get information about the event. You get handle about this structure from the *snd_pcm_event_t()* structure.

The members include:

flags

The previous state that the channel had transitioned from, which can be one of the following list of states. For more information about the various states, see the "*PCM state machine*" section in the Audio Architecture chapter of this book.

new_status

The current state that the channel had transitioned to, which can be one of the following list of states. For more information about the various states, see the "*PCM state machine*" section in the Audio Architecture chapter of this book.

flags

A flag that indicates more information about the new status. It can be one of the following values:

- SND_PCM_STATUS_EVENT_HARD_SUSPEND—The channel is in a hard suspended (SND_PCM_STATUS_SUSPENDED) state, which means the channel can't start playing when you call a `snd_pcm_*_resume()` function.
- SND_PCM_STATUS_EVENT_SOFT_SUSPEND— The channel is in a soft suspended (SND_PCM_STATUS_SUSPENDED) state, which means the channel can start playing when you call a `snd_pcm_*_resume()` function.
- SND_PCM_STATUS_EVENT_AUTOPAUSE—The channel is in the SUSPENDED (SND_PCM_STATUS_SUSPENDED) state, but when the suspension is lifted, the channel

transitions to the PAUSED (SND_PCM_STATUS_PAUSED) state because the audio concurrency management policies were applied. In other words, the application didn't call a `snd_pcm_*_pause()` function to move to the PAUSED state.

.

Classification:

QNX Neutrino

Related Links

snd_pcm_channel_read_event() (p. 274)
  Get a PCM event

`snd_pcm_event_t` (p. 295)
  Information about the PCM event that occurred

snd_pcm_t

The handle for the PCM device.

Synopsis:

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

typedef struct snd_pcm snd_pcm_t;
```

Library:

libasound.so

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

Description:

This data type is an opaque data type that is used to work with PCM card and PCM devices.

You should use one handle for each thread to ensure that functions run thread safe. Avoid using the same handle across multiple threads.

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

Library:

libasound.so

Use the `-l asound` option with `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	Read the Caveats

Caveats:

This function is not thread safe if *pcm*(`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_open() (p. 347)
: *Create a handle and open a connection to a specified audio interface*

snd_pcm_open_preferred() (p. 353)
: *Create a handle and open a connection to the preferred audio interface*

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;
```

Library:

libasound.so

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

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

Related Links

snd_pcm_plugin_get_voice_conversion() (p. 374)
Get the current voice conversion structure for a channel

snd_pcm_plugin_set_voice_conversion() (p. 399)
Set the current voice conversion structure for a channel

"Controlling voice conversion" (p. 36)

Configuration of the **libasound** library is based on the maximum number of voices supported in hardware.

in the Playing and Capturing Audio Data chapter

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()*, *snd_pcm_open()*, or *snd_pcm_open_preferred()*.

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 with `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()* or *snd_pcm_playback_prepare()*.

This function may suspend a process if blocking behavior is active (see *snd_pcm_nonblock_mode()*) 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` 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. When this function was called, it was in the SND_PCM_STATUS_PAUSED or SND_PCM_STATUS_SUSPENDED state.

-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` structure.

-EWOULDBLOCK

The write would have blocked (nonblocking write).

Classification:

QNX Neutrino

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Read the Caveats

Caveats:

This function is not thread safe if *handle* (`snd_pcm_t`) is used across multiple threads.

Related Links

snd_pcm_channel_prepare() (p. 272)
Signal the driver to ready the specified channel

snd_pcm_playback_prepare() (p. 367)
: *Signal the driver to ready the playback channel*

snd_pcm_plugin_write() (p. 407)
: *Transfer PCM data to playback channel (plugin-aware)*

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 with `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
Interrupt handler	No
Signal handler	Yes

Safety:	
Thread	Yes

Related Links

in the QNX Neutrino *C Library Reference*

snd_switch_t

Information about a mixer's switch

Synopsis:

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

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
            {
                struct
                {
                    uint16_t input[15];
                    uint16_t output[15];
                    uint16_t cnt;    /* Read-only (configured by driver); defines
                                        the number of concurrent routes supported
                                        through the switch */
                } active;

                /* The below items are read-only (configured by driver) to
                   communicate the inputs and outputs connected to the switch */
                uint16_t inputs[15];
                uint16_t inputs_cnt;
                uint16_t outputs[15];
                uint16_t outputs_cnt;
            }
            routing_list;

            struct
            {
                uint16_t selected_items[30];
                uint16_t selected_items_cnt;
                uint16_t items[30];
                uint16_t items_cnt;
            }
            multi_selection_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()*.

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:

Type	Union member	Description
SND_SW_TYPE_BOOLEAN	*enable*	A simple on and off switch
SND_SW_TYPE_BYTE	*byte*	An 8-bit value constrained between a minimum and maximum setting
SND_SW_TYPE_DWORD	*dword*	A 32-bit value constrained between a minimum and maximum setting
SND_SW_TYPE_LIST	*list*	A 32-bit value selected from a list of values. The *items_cnt* argument is the number of valid items in the array.
SND_SW_TYPE_MULTI_SEL_LIST	*multi_selection_list*	A multiselection list switch; see below.
SND_SW_TYPE_ROUTING_LIST	*routing_list*	A routing list switch; see below.
SND_SW_TYPE_STRING_11	*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.
SND_SW_TYPE_WORD	*word*	A 16-bit value constrained between a minimum and maximum setting

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.

SND_SW_TYPE_ROUTING_LIST and SND_SW_TYPE_MULTI_SEL_LIST are similar:

- A *routing list switch* allows for up to 15 input and 15 output pins/connections and up to 15 active/concurrent routes. This switch type holds all of the available/allowable inputs and outputs that can be controlled by the switch, and the currently active/selected routes (input-output pairs).

  This allows the client application to know what it can and can't do with any specific instance of the switch. It also makes it easy to query which inputs and outputs are currently selected/active. Since each instance of this switch type is coded with a specific set of inputs and output identifiers, it also makes it easy for the driver code to validate the client app's switch configuration request and to apply the client's switch configuration.

  The *active.cnt* is read-only and is set by the driver code. It describes the number of concurrent routes supported by the particular instance of the switch. The client and driver should use an index only up to *active.cnt* – 1 when reading or writing to the active structure of the *routing_list* switch.

- A *multiselection list switch* is similar to a routing list switch, in that all available/allowable inputs are coded into the switch (up to 30 inputs/items). As mentioned above, this allows the client to query the switch to know exactly what inputs can be selected and what inputs are currently selected.

  This also makes it easy for the driver to do error checking on the client's configuration requests. This switch type allows the client to select multiple inputs (there's an array of available inputs/items and an array of selected inputs/items) to be routed to a single output (i.e., hardware mixing).

Classification:

QNX Neutrino

Related Links

snd_ctl_mixer_switch_read() (p. 160)
Get a mixer switch setting

snd_ctl_mixer_switch_write() (p. 162)
Adjust a mixer switch setting

Appendix A
afm_ctl.c example

This is the source for the afm_ctl utility.

The afm_ctl utility is shipped with Acoustic Management Platform 3.0.

For information about using this utility, see afm_ctl in the *QNX Neutrino Utilities Reference*.

```
/*
 * Copyright 2016, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd.  (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/asoundlib.h>
#include <string.h>
#include <devctl.h>
#include <errno.h>
#include <ctype.h>
#include <ioctl.h>


//*****************************************************************************
/* *INDENT-OFF* */
#ifdef __USAGE
%C [Options]

Options:
    -a [card#:]<dev#>        the AFM card & device number to start/issue command
                             OR
    -a [name]                the AFM card name (e.g. voice, icc) to start/issue command

    -f <filename>            set wav file (full path) (recorder/player AFMs only)
    -m <mode>                set audio mode
    -c                       reset audio mode
    -t <dataset>             load runtime acoustic processing dataset
    -u                       clear runtime acoustic processing dataset
    -l <ms_offset>           start microphone latency test
    -r <ms_offset>           start reference latency test
    -v <rpm>                 set rpm VIN value
                             (diagnostic use only - only applicable if rpm is a base VIN)
    -x <param_id:chn>[:data] calls set/get data on afm for ap parameters of size int16_t
    -y <param_id:chn>[:data] calls set/get data on afm for ap parameters of size int32_t
    -z <param_id>[:data]     calls set/get data on afm for afm parameters of size int32_t
    -s                       stop AFM
#endif
```

```
/* *INDENT-ON* */
//***************************************************************************

const char* optstring = "a:f:m:sl:r:cv:t:ux:y:z:";
#define LATENCY_TEST_MIC 0
#define LATENCY_TEST_REF 1

static snd_afm_t* setup_afm_handle(char* arg)
{
    int rtn;
    int dev = 0, card = 0;
    const char* name = NULL;
    snd_afm_t* afm_handle = NULL;

    if (strchr(arg, ':'))
    {
        card = atoi(arg);
        dev = atoi(strchr(arg, ':') + 1);
    }
    else if (isalpha(arg[0]) || (arg[0] == '/'))
        name = arg;
    else
        dev = atoi(arg);

    if (name)
    {
        printf("Using %s\n", name);
        rtn = snd_afm_open_name(&afm_handle, name);
    }
    else
    {
        printf("Using card %d device %d \n", card, dev);
        rtn = snd_afm_open(&afm_handle, card, dev);
    }
    if (rtn != EOK)
    {
        fprintf(stderr, "snd_afm_open failed: (%d) %s\n", rtn, snd_strerror(rtn));
        return NULL;
    }

    return afm_handle;
}

static int latency_test(snd_afm_t* afm_handle, int input_device, int ms_offset)
{
    int rtn = EOK;
    int fd;
    snd_afm_latency_test_t test;

    if ((fd = snd_afm_file_descriptor(afm_handle)) > 0)
    {
        test.input_device = input_device;  /* 0 = mic, 1 = ref */
        test.input_voice = 0;
        test.ms_offset = ms_offset;

        if (ioctl(fd, SND_AFM_IOCTL_START_LATENCY_TEST, &test) < 0) {
            printf("Failed to start latency test: (%d) %s\n", errno, strerror(errno));
            rtn = -errno; /* Negate errno to match snd_xxx error codes */
        }
    }
    else
    {
        printf("Failed to get AFM descriptor for latency test (%d) %s\n", errno, strerror(errno));
        rtn = -errno; /* Negate errno to match snd_xxx error codes */
    }
    return rtn;
}

static int set_audio_mode(snd_afm_t* afm_handle, const char* mode)
{
    int rtn;
    char str[64];

    if (mode[0]) {
        printf("Setting mode to %s\n", mode);
    } else {
        printf("Clearing mode\n");
    }
    if ((rtn = snd_afm_set_audio_mode(afm_handle, mode)) != EOK)
        printf("Failed to set mode: (%d) %s\n", rtn, snd_strerror(rtn));

    if ((rtn = snd_afm_get_audio_mode(afm_handle, str, sizeof(str))) != EOK)
        printf("Failed to get mode: (%d) %s\n", rtn, snd_strerror(rtn));
    else
        printf("Audio Mode = %s\n", str);
    return rtn;
}

static int set_dataset(snd_afm_t* afm_handle, const char* dataset)
{
    int rtn;
    int ap_status = 0;

    if (dataset[0])
        printf("Loading dataset %s\n", dataset);
     else
        printf("Clearing dataset\n");

    if ((rtn = snd_afm_load_ap_dataset(afm_handle, dataset, &ap_status)) != EOK)
        printf("Failed to set dataset: (%d) %s\n", rtn, snd_strerror(rtn));
    if (ap_status != 0)
        printf("Acoustic processing returned status=0x%04X\n", ap_status);
    return rtn;
}

static int set_rpm_vin(snd_afm_t* afm_handle, int rpm)
{
    int rtn;
    int vinCount = 0;
    if ((rtn = snd_afm_get_vin_list_count(afm_handle, &vinCount)) == EOK) {
        snd_afm_vin_list_item_t* vin_items = alloca( sizeof(snd_afm_vin_list_item_t) * vinCount);
        snd_afm_vin_pair_t* vin_pairs = alloca( sizeof(snd_afm_vin_pair_t) * vinCount);
        if (vin_items && vin_pairs)
        {
            memset(vin_pairs, 0, sizeof(snd_afm_vin_pair_t) * vinCount);
            if ((rtn = snd_afm_get_vin_list(afm_handle, vin_items, vinCount)) == EOK) {
                int i;
                for (i=0; i<vinCount; i++) {
                    vin_pairs[i].key = vin_items[i].key;
                    if (vin_items[i].is_rpm) {
                        vin_pairs[i].value = rpm;
                        printf("Vin 0x%X set to %d\n", vin_pairs[i].key,  vin_pairs[i].value);
                    }
                }
                rtn = snd_afm_set_vin_stream(afm_handle, vin_pairs, vinCount);
            }
        } else {
            rtn = -errno; /* Negate errno to match snd_xxx error codes */
        }
    }
    if (rtn != EOK)
    {
        printf("Failed to set RPM (%d) %s\n", rtn, snd_strerror(rtn));
    }
    return rtn;
}

/* Set/get acoustic processing parameter */
static int set_ap_data(snd_afm_t* afm_handle, char* arg, size_t data_size)
{
    int set_data, rtn;
    int32_t data;
    snd_afm_ap_param_t param = {
        .size = data_size,
    };

    param.dataId = strtol(arg, &arg, 0);
    if (errno == ERANGE || errno == EINVAL) {
        fprintf(stderr, "Invalid ap data id\n");
        return 1;
    }
    if (*arg) arg++;
    param.channel = strtol(arg, &arg, 0);
    if (errno == ERANGE || errno == EINVAL) {
        fprintf(stderr, "Invalid channel\n");
        return 1;
    }
    if (*arg) arg++;
    data = strtol(arg, &arg, 0);
    set_data = (errno != ERANGE && errno != EINVAL);
    errno = EOK;
    if (set_data) {
        printf("AFM set_ap_data id=0x%04x data=%d ", param.dataId, data);
        rtn = snd_afm_set_ap_data(afm_handle, &param, &data);
    } else {
        printf("AFM get_ap_data id=0x%04x ", param.dataId);
        rtn = snd_afm_get_ap_data(afm_handle, &param, &data);
    }
    printf("ret=%d ap_ret=%d data_ret=%d -- %s\n", rtn, param.status, data, strerror(-rtn));
    return rtn;
}


/* Set/get acoustic processing parameter */
static int set_afm_param(snd_afm_t* afm_handle, char* arg, size_t data_size)
{
    int set_data, rtn;
    int32_t data, param_id;

    param_id = strtol(arg, &arg, 0);
    if (errno == ERANGE || errno == EINVAL) {
        fprintf(stderr, "Invalid afm param data id\n");
        return 1;
    }
    if (*arg) arg++;
    data = strtol(arg, &arg, 0);
    set_data = (errno != ERANGE && errno != EINVAL);
    errno = EOK;
    if (set_data) {
        printf("AFM set_param id=0x%04x data=%d ", param_id, data);
        rtn = snd_afm_set_param(afm_handle, param_id, data_size, &data);
    } else {
        printf("AFM get_param id=0x%04x ", param_id);
        rtn = snd_afm_get_param(afm_handle, param_id, &data_size, &data);
    }
    printf("ret=%d data_ret=%d -- %s\n", rtn, data, strerror(-rtn));
    return rtn;
}


int main(int argc, char *argv[])
{
    int rtn = EOK;
    snd_afm_t *afm_handle = NULL;
    int start_afm = 1;
    int c;

    /* first setup afm handle */
    while ((c = getopt(argc, argv, optstring)) != EOF)
    {
        switch (c)
        {
        case 'a':
            if (!(afm_handle = setup_afm_handle(optarg))) {
                return EXIT_FAILURE;
            }
            break;
        default:
            /* handle other options instead of starting AFM */
            start_afm = 0;
            break;
        }
    }

    if (!afm_handle) {
        printf("No AFM specified, use -a option\n");
        return EXIT_FAILURE;
    }
    if (start_afm) {
        printf("Starting AFM \n");
        if ((rtn = snd_afm_start(afm_handle)) != EOK)
            printf("Failed to start AFM: (%d) %s\n", rtn, snd_strerror(rtn));

        snd_afm_close(afm_handle);
        return (rtn == EOK) ? EXIT_SUCCESS : EXIT_FAILURE;
    }

    /* reset optind and handle other options */
    optind = 1;
    while (((c = getopt(argc, argv, optstring)) != EOF) && (rtn == EOK))
    {
        switch (c)
        {
        case 'a':
            break;

        case 's':
            printf("Stopping AFM \n");
            if ((rtn = snd_afm_stop(afm_handle)) != EOK)
                printf("Failed to stop AFM: (%d) %s\n", rtn, snd_strerror(rtn));
            break;

        case 'l':
            rtn = latency_test(afm_handle, LATENCY_TEST_MIC, atoi(optarg));
            break;

        case 'r':
            rtn = latency_test(afm_handle, LATENCY_TEST_REF, atoi(optarg));
            break;

        case 'f':
            printf("Setting filename %s, len = %zu\n", optarg, strlen(optarg));
            if ((rtn = snd_afm_set_path(afm_handle, SND_AFM_WAV_FILE, optarg)) != EOK)
                printf("Failed to set filename: (%d) %s\n", rtn, snd_strerror(rtn));
            break;

        case 'm':
            rtn = set_audio_mode(afm_handle, optarg);
            break;

        case 'c':
            rtn = set_audio_mode(afm_handle, "");
            break;

        case 't':
            rtn = set_dataset(afm_handle, optarg);
            break;

        case 'u':
            rtn = set_dataset(afm_handle, "");
            break;

        case 'v':
            rtn = set_rpm_vin(afm_handle, strtoul(optarg, NULL, 0));
            break;

        case 'x':
            rtn = set_ap_data(afm_handle, optarg, sizeof(uint16_t));
            break;

        case 'y':
            rtn = set_ap_data(afm_handle, optarg, sizeof(uint32_t));
            break;

        case 'z':
            rtn = set_afm_param(afm_handle, optarg, sizeof(uint32_t));
            break;

        default:
            fprintf(stderr, "Invalid option '%c'\n", c);
            rtn = -1;
            break;
        }
    }
```

```
    snd_afm_close(afm_handle);

    return (rtn == EOK) ? EXIT_SUCCESS : EXIT_FAILURE;
}

#if defined(__QNXNTO__) && defined(__USESRCVERSION)
#include <sys/srcversion.h>
__SRCVERSION("$URL$ $Rev$")
#endif
```

Appendix B
apx_ctl.c example

This is the source for the apx_ctl utility.

The apx_ctl utility is shipped with Acoustic Management Platform 3.0.

For information about using this utility, see apx_ctl in the *QNX Neutrino Utilities Reference*.

```
/*
 * Copyright 2018, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd.  (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/asoundlib.h>
#include <string.h>
#include <devctl.h>
#include <errno.h>
#include <ctype.h>
#include <ioctl.h>


//*****************************************************************************
/* *INDENT-OFF* */
#ifdef __USAGE
%C [Options]

Options:
    -a [card#:]<dev#>                     the card & device number to access
                                            OR
    -a[name]                              the card name to access
    -e <chns:volume_chn0:...:volume_chnN> set volume of external amplifiers (outside of io-audio system)
    -u <user volume>                      override default user volume, with system output volume
    -t <apx_id:dataset>                   load runtime acoustic processing dataset
    -x <apx_id:param_id:chn>[:data]       calls set/get data on apx for parameters of size int16_t
    -y <apx_id:param_id:chn>[:data]       calls set/get data on apx for parameters of size int32_t
                                          SPM : apx_id=1
                                          SFO : apx_id=2
#endif
/* *INDENT-ON* */
//*****************************************************************************

const char* optstring = "a:e:u:t:x:y:";

static snd_pcm_t* setup_pcm_handle(char* arg)
{
    int rtn;
    int dev = 0, card = 0;
    const char* name = NULL;
    snd_pcm_t* pcm_handle = NULL;

    if (strchr (arg, ':'))
    {
        card = strtol (arg, &arg, 0);
        if (errno == ERANGE || errno == EINVAL) {
            fprintf(stderr, "Invalid card number\n");
            return NULL;
        }
        if (*arg)  arg++;
        dev = strtol (arg, NULL, 0);
        if (errno == ERANGE || errno == EINVAL) {
            fprintf(stderr, "Invalid device number\n");
            return NULL;
        }
    } else if (isalpha (arg[0]) || arg[0] == '/') {
        name = arg;
    } else {
        dev = strtol (arg, NULL, 0);
        if (errno == ERANGE || errno == EINVAL) {
            fprintf(stderr, "Invalid device number\n");
            return NULL;
        }
    }

    if (name) {
        fprintf (stderr, "Using device %s\n", name);
        rtn = snd_pcm_open_name(&pcm_handle, name, SND_PCM_OPEN_PLAYBACK);
    } else {
        fprintf (stderr, "Using card %d device %d \n", card, dev);
        rtn = snd_pcm_open (&pcm_handle, card, dev, SND_PCM_OPEN_PLAYBACK);
    }
    if (rtn < 0) {
        fprintf(stderr, "Cannot open pcm device\n");
        return NULL;
    }
    return pcm_handle;
}

static int set_ext_vol(snd_pcm_t* pcm_handle, const char* arg)
{
    int i, rtn;
    int16_t ext_vol [32] = {0};
    int ext_vol_channels = strtol (optarg, &optarg, 0);
    if (errno == ERANGE || errno == EINVAL || ext_vol_channels > 32) {
        fprintf(stderr, "Invalid number of external volume channels\n");
        return 1;
    }
    for (i = 0; (i < 32) && optarg && (*optarg == ':') && (errno != ERANGE) && (errno != EINVAL); i++) {
        optarg++;
        ext_vol[i] = strtol (optarg, &optarg, 0);
    }
    rtn = snd_pcm_set_apx_external_volume(pcm_handle, ext_vol_channels, ext_vol);
    printf("APX set external volume ret=%d -- %s\n", rtn, strerror(rtn));
    return rtn;
}

static int set_user_vol(snd_pcm_t* pcm_handle, int16_t vol)
```

```
{
    int rtn = snd_pcm_set_apx_user_volume(pcm_handle, vol);
    printf("APX set user volume to %d ret=%d -- %s\n", vol, rtn, strerror(rtn));
    return rtn;
}

static int set_dataset(snd_pcm_t* pcm_handle, char* dataset)
{
    int rtn, ap_status = 0;
    int32_t apx_id = strtol (dataset, &dataset, 0);
    if (errno == ERANGE || errno == EINVAL) {
        fprintf(stderr, "Invalid apx id\n");
        return 1;
    }
    dataset++;
    printf("Loading dataset %s\n", dataset);
    if ((rtn = snd_pcm_load_apx_dataset(pcm_handle, apx_id, dataset, &ap_status)) != EOK)
        printf("Failed to load dataset: (%d) %s\n", rtn, snd_strerror(rtn));
    if (ap_status != 0) {
        printf("Acoustic processing returned status=0x%04X\n", ap_status);
    }
    return rtn;
}

static int set_ap_data(snd_pcm_t* pcm_handle, char* arg, size_t data_size)
{
    int set_data, rtn;
    int32_t apx_id;
    uint32_t data;
    snd_pcm_apx_param_t param = {
        .size = data_size,
    };

    apx_id = strtol (arg, &arg, 0);
    if (errno == ERANGE || errno == EINVAL) {
        fprintf(stderr, "Invalid apx id\n");
        return 1;
    }
    if (*arg) arg++;
    param.dataId = strtol (arg, &arg, 0);
    if (errno == ERANGE || errno == EINVAL) {
        fprintf(stderr, "Invalid apx data id\n");
        return 1;
    }
    if (*arg) arg++;
    param.channel = strtol (arg, &arg, 0);
    if (errno == ERANGE || errno == EINVAL) {
        fprintf(stderr, "Invalid apx channel\n");
        return 1;
    }
    if (*arg) arg++;
    data = strtol (arg, NULL, 0);
    set_data = (errno != ERANGE && errno != EINVAL);
    errno = EOK;
    if (set_data) {
        printf("APX set_ap_data apx=%d id=0x%04x data=%d ", apx_id, param.dataId, data);
        rtn = snd_pcm_set_apx_data(pcm_handle, apx_id, &param, &data);
    } else {
        printf("APX get_ap_data apx=%d id=0x%04x ", apx_id, param.dataId);
        rtn = snd_pcm_get_apx_data(pcm_handle, apx_id, &param, &data);
    }
    printf("ret=%d ap_ret=%d data_ret=%d -- %s\n", rtn, param.status, data, strerror(-rtn));
    return rtn;
}

int main(int argc, char *argv[])
{
    int rtn = EOK;
    snd_pcm_t *pcm_handle = NULL;
    int c;

    /* first setup pcm handle */
    while ((c = getopt(argc, argv, optstring)) != EOF)
    {
        switch (c)
        {
        case 'a':
            if (!(pcm_handle = setup_pcm_handle(optarg))) {
                return EXIT_FAILURE;
            }

            break;
        default:
            break;
        }
    }

    if (!pcm_handle) {
        fprintf (stderr, "Using preferred device \n");
        if (snd_pcm_open_preferred(&pcm_handle, NULL, NULL, SND_PCM_OPEN_PLAYBACK) != EOK)  {
            fprintf(stderr, "Cannot open pcm device\n");
            return EXIT_FAILURE;
        }
    }

    /* reset optind and handle other options */
    optind = 1;
    while (((c = getopt(argc, argv, optstring)) != EOF) && (rtn == EOK))
    {
        switch (c)
        {
        case 'a':
            break;

        case 'e':
            rtn = set_ext_vol(pcm_handle, optarg);
            break;

        case 'u':
            rtn = set_user_vol(pcm_handle, atoi(optarg));
            break;

        case 't':
            rtn = set_dataset(pcm_handle, optarg);
            break;

        case 'x':
            rtn = set_ap_data(pcm_handle, optarg, sizeof(uint16_t));
            break;

        case 'y':
            rtn = set_ap_data(pcm_handle, optarg, sizeof(uint32_t));
            break;

        default:
            printf("Invalid option '%c'\n", c);
            rtn = -1;
            break;
        }
    }
    snd_pcm_close(pcm_handle);
    return (rtn == EOK) ? EXIT_SUCCESS : EXIT_FAILURE;
}

#if defined(__QNXNTO__) && defined(__USESRCVERSION)
#include <sys/srcversion.h>
__SRCVERSION("$URL$ $Rev$")
#endif
```

Appendix C
`wave.c` example

This is a sample application that plays back audio data.

For information about using this utility, see "*wave*" in the *QNX Neutrino Utilities Guide*.

```
/*
 * $QNXLicenseC:
 * Copyright 2016, 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 <sys/slog2.h>
#include <time.h>
#include <ctype.h>
#include <limits.h>
#include <signal.h>
#include <pthread.h>

#include <sys/asoundlib.h>

#define WRITE_RETRY_TIMES 5

typedef struct
{
    char    tag[4];
    int32_t length;
}
RiffTag;

typedef struct
{
    char     Riff[4];
    int32_t  Size;
    char     Wave[4];
}
RiffHdr;

typedef struct
{
    int16_t   FormatTag;
    int16_t   Channels;
    int32_t   SamplesPerSec;
    int32_t   AvgBytesPerSec;
    int16_t   BlockAlign;
    int16_t   BitsPerSample;
}
FmtChunk;

typedef struct
{
    FILE *file1;
    struct timespec start_time;
}
WriterData;

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;
int use_writer_thread = 0;
useconds_t frag_period_us;
snd_mixer_group_t group;
snd_mixer_t *mixer_handle = NULL;
snd_pcm_t *pcm_handle = NULL;
snd_pcm_channel_params_t pp;
char *mSampleBfr1 = NULL;
unsigned int mDataSize;
bool mBigEndian = false;
int nonblock = 0;
int repeat = 0;
long int data_position = 0;
int stdin_raw = 0;
WriterData wd;

static slog2_buffer_t slog_handle;
static slog2_buffer_set_config_t slog_config;

static 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);
}


static int
CheckHdr (FILE * fp)
{
    RiffHdr riffHdr = { "", 0 };

    if (fread ((unsigned char *) &riffHdr, sizeof (RiffHdr), 1, fp) == 0)
        return -1;

    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;
}


static 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));
}

static 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));
}

static void
cleanup(void)
{
    if (stdin_raw)
        dev_unraw (fileno (stdin));
    if (mixer_handle)
        snd_mixer_close (mixer_handle);
    if (pcm_handle)
        snd_pcm_close (pcm_handle);
    if (wd.file1)
        fclose(wd.file1);
    if (mSampleBfr1)
        free(mSampleBfr1);
}

static void
cleanup_and_exit(int exit_code)
{
 cleanup();
 exit(exit_code);
}

static 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 );
            return;
        case 'r':
            snd_pcm_playback_resume( pcm_handle );
            return;
        case 'i':
            {
                char buf[100] = {0};
                snd_pcm_channel_status_t status;

                memset (&status, 0, sizeof (status));
                status.channel = SND_PCM_CHANNEL_PLAYBACK;
                if ((rtn = snd_pcm_plugin_status (pcm_handle, &status)) < 0)
                {
                    fprintf (stderr, "plugin_status: playback channel status error\n");
                    return;
                }

                /* Display Subchn state */
                printf("\nSubchn State = ");
                switch (status.status)
                {
                    case SND_PCM_STATUS_NOTREADY:
                        printf("NOTREADY\n");
                        break;
                    case SND_PCM_STATUS_READY:
                        printf("READY\n");
                        break;
                    case SND_PCM_STATUS_PREPARED:
                        printf("PREPARED\n");
                        break;
                    case SND_PCM_STATUS_RUNNING:
                        printf("RUNNING\n");
                        break;
                    case SND_PCM_STATUS_PAUSED:
                        printf("PAUSED\n");
                        break;
                    case SND_PCM_STATUS_SUSPENDED:
                        printf("SUSPENDED\n");
                        break;
                    case SND_PCM_STATUS_UNDERRUN:
                        printf("UNDERRUN\n");
                        break;
                    case SND_PCM_STATUS_OVERRUN:
                        printf("OVERRUN\n");
                        break;
                    case SND_PCM_STATUS_CHANGE:
                        printf("CHANGE\n");
                        break;
                    case SND_PCM_STATUS_ERROR:
                        printf("ERROR\n");
                        break;
                    default:
                        printf("UNKNOWN\n");
                        break;
                }
                /* Display Ducking State */
                if (status.ducking_state & SND_PCM_DUCKING_STATE_FORCED_ACTIVE)
                {
                    strcat(buf, "FORCED_ACTIVE");
                    if (status.ducking_state & SND_PCM_DUCKING_STATE_ACTIVE)
                        strcat(buf, "|ACTIVE");
               }
               else if (status.ducking_state & SND_PCM_DUCKING_STATE_ACTIVE)
                   strcat(buf, "ACTIVE");
               else
                   strcat(buf, "INACTIVE");
               if (status.ducking_state & SND_PCM_DUCKING_STATE_HARD_SUSPENDED)
                   strcat(buf, "|HARD_SUSPENDED");
               if (status.ducking_state & SND_PCM_DUCKING_STATE_SOFT_SUSPENDED)
                   strcat(buf, "|SOFT_SUSPENDED");
               if (status.ducking_state & SND_PCM_DUCKING_STATE_PAUSED)
                   strcat(buf, "|AUTOPAUSED");
```

```
            printf("Ducking State = %s\n", buf);
            /* Display buffer statistics */
            printf("scount = %d, used = %d, free = %d, underrun = %d, overrun = %d\n",
                   status.scount, status.count, status.free, status.underrun, status.overrun);
         }
         return;
     case 't':
         if (repeat) {
             repeat = 0;
             printf("Stop to repeat playing\n");
             return;
         } else
             break;
     // Exit the program
     case 3: // Ctrl-C
     case 27: // Escape
         running = false;
         return;
     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;
 }

 /* Adjust the volume by 10 or by the remaining volume steps until the min or max (whichever is smaller) */
 switch (c)
 {
     case 'q':
         if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
             group.volume.names.front_left += min(group.max - group.volume.names.front_left, 10);
         if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
             group.volume.names.rear_left += min(group.max - group.volume.names.rear_left, 10);
         if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
             group.volume.names.woofer += min(group.max - group.volume.names.woofer, 10);
         break;
     case 'a':
         if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
             group.volume.names.front_left -= min(group.volume.names.front_left - group.min, 10);
         if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
             group.volume.names.rear_left -= min(group.volume.names.rear_left - group.min, 10);
         if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
             group.volume.names.woofer -= min(group.volume.names.woofer - group.min, 10);
         break;
     case 'w':
         if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
             group.volume.names.front_left += min(group.max - group.volume.names.front_left, 10);
         if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
             group.volume.names.rear_left += min(group.max - group.volume.names.rear_left, 10);
         if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
             group.volume.names.front_center += min(group.max - group.volume.names.front_center, 10);
         if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
             group.volume.names.front_right += min(group.max - group.volume.names.front_right, 10);
         if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
```

```
                group.volume.names.rear_right += min(group.max - group.volume.names.rear_right, 10);
            if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
                group.volume.names.woofer += min(group.max - group.volume.names.woofer, 10);
            break;
        case 's':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_LEFT)
                group.volume.names.front_left -= min(group.volume.names.front_left - group.min, 10);
            if (group.channels & SND_MIXER_CHN_MASK_REAR_LEFT)
                group.volume.names.rear_left -= min(group.volume.names.rear_left - group.min, 10);
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
                group.volume.names.front_center -= min(group.volume.names.front_center - group.min, 10);
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
                group.volume.names.front_right -= min(group.volume.names.front_right - group.min, 10);
            if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
                group.volume.names.rear_right -= min(group.volume.names.rear_right - group.min, 10);
            if (group.channels & SND_MIXER_CHN_MASK_WOOFER)
                group.volume.names.woofer -= min(group.volume.names.woofer - group.min, 10);
            break;
        case 'e':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
                group.volume.names.front_right += min(group.max - group.volume.names.front_right, 10);
            if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
                group.volume.names.rear_right += min(group.max - group.volume.names.rear_right, 10);
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
                group.volume.names.front_center += min(group.max - group.volume.names.front_center, 10);
            break;
        case 'd':
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_RIGHT)
                group.volume.names.front_right -= min(group.volume.names.front_right - group.min, 10);
            if (group.channels & SND_MIXER_CHN_MASK_REAR_RIGHT)
                group.volume.names.rear_right -= min(group.volume.names.rear_right - group.min, 10);
            if (group.channels & SND_MIXER_CHN_MASK_FRONT_CENTER)
                group.volume.names.front_center -= min(group.volume.names.front_center - group.min, 10);
            break;
    }

    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);
    }
}

static void handle_mixer()
{
    fd_set  rfds;
    int mixer_fd = snd_mixer_file_descriptor (mixer_handle);

    FD_ZERO(&rfds);
    FD_SET ( mixer_fd, &rfds);

    if (select (mixer_fd + 1, &rfds, NULL, NULL, NULL) == -1)
        perror ("select");

    snd_mixer_callbacks_t callbacks = { 0, 0, 0, 0 };

    snd_mixer_read (mixer_handle, &callbacks);
}

static void display_status_event (snd_pcm_event_t *event)
{
 char flag_buf[100] = {0};

 if (event->data.audiomgmt_status.flags & SND_PCM_STATUS_EVENT_HARD_SUSPEND)
 {
  if (event->data.audiomgmt_status.flags & ~(SND_PCM_STATUS_EVENT_HARD_SUSPEND|(SND_PCM_STATUS_EVENT_HARD_SUSPEND - 1U)))
   strcat(flag_buf, "HARD_SUSPEND|");
  else
   strcat(flag_buf, "HARD_SUSPEND");
 }

 if (event->data.audiomgmt_status.flags & SND_PCM_STATUS_EVENT_SOFT_SUSPEND)
 {
  if (event->data.audiomgmt_status.flags & ~(SND_PCM_STATUS_EVENT_SOFT_SUSPEND|(SND_PCM_STATUS_EVENT_SOFT_SUSPEND-1U)))
   strcat(flag_buf, "SOFT_SUSPEND|");
  else
   strcat(flag_buf, "SOFT_SUSPEND");
 }

 if (event->data.audiomgmt_status.flags & SND_PCM_STATUS_EVENT_AUTOPAUSE)
 {
  if (event->data.audiomgmt_status.flags & ~(SND_PCM_STATUS_EVENT_AUTOPAUSE | (SND_PCM_STATUS_EVENT_AUTOPAUSE - 1U)))
   strcat(flag_buf, "AUTOPAUSE|");
  else
   strcat(flag_buf, "AUTOPAUSE");
 }

 if (event->data.audiomgmt_status.flags == 0)
 {
  strcat(flag_buf, "None");
 }

 switch (event->data.audiomgmt_status.new_status)
 {
  case SND_PCM_STATUS_SUSPENDED:
   printf("Audio Management Status event received - SUSPENDED\n");
   printf("\tFlags = (0x%x) %s\n", event->data.audiomgmt_status.flags, flag_buf);
   break;
  case SND_PCM_STATUS_RUNNING:
   printf("Audio Management Status event received - RUNNING\n");
   printf("\tFlags = (0x%x) %s\n", event->data.audiomgmt_status.flags, flag_buf);
   break;
  case SND_PCM_STATUS_PAUSED:
   printf("Audio Management Status event received - PAUSED\n");
   printf("\tFlags = (0x%x) %s\n", event->data.audiomgmt_status.flags, flag_buf);
   break;
  default:
   break;
 }
}

static void display_mute_event (snd_pcm_event_t *event)
{
    char flag_buf[100] = {0};

    if (event->data.audiomgmt_mute.mute == 1)
    {
        if (event->data.audiomgmt_mute.reason & SND_PCM_MUTE_EVENT_HIGHER_PRIORITY)
        {
            if (event->data.audiomgmt_mute.reason & ~(SND_PCM_MUTE_EVENT_HIGHER_PRIORITY))
                strcat(flag_buf, "MUTE_BY_HIGHER|");
            else
                strcat(flag_buf, "MUTE_BY_HIGHER");
        }
        if (event->data.audiomgmt_mute.reason & SND_PCM_MUTE_EVENT_SAME_PRIORITY)
            strcat(flag_buf, "MUTE_BY_SAME");

        printf("Audio Management Mute event received - Muted by %s\n", flag_buf);
    }
    else
    {
        printf("Audio Management Mute event received - Un-Muted\n");
    }
}

static void handle_pcm_events()
{
    fd_set  ofds;
    snd_pcm_event_t event;
    int rtn = EOK;
    int pcm_fd = snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_PLAYBACK);

    FD_ZERO(&ofds);
    FD_SET ( pcm_fd, &ofds);

    if (select (pcm_fd + 1, NULL, NULL, &ofds, NULL) == -1)
        perror ("select");

    if ((rtn = snd_pcm_channel_read_event (pcm_handle, SND_PCM_CHANNEL_PLAYBACK, &event)) == EOK)
    {
        switch (event.type)
        {
            case SND_PCM_EVENT_AUDIOMGMT_STATUS:
                display_status_event(&event);
                break;
            case SND_PCM_EVENT_AUDIOMGMT_MUTE:
                display_mute_event(&event);
                break;
            case SND_PCM_EVENT_OUTPUTCLASS:
                printf("Output class event received - output class changed from %d to %d\n",
                       event.data.outputclass.old_output_class, event.data.outputclass.new_output_class);
                break;
            case SND_PCM_EVENT_UNDERRUN:
                printf("Underrun event received\n");
                break;
            default:
                printf("Unknown PCM event type - %d\n", event.type);
                break;
        }
    }
    else
        printf("snd_pcm_channel_read_event() failed with %d\n", rtn);

}

static void write_audio_data(WriterData *wd)
{
    struct timespec current_time;
    snd_pcm_channel_status_t status;
    int written = 0;
    int retries = 0;
    int remainder = mDataSize - N;

    if (repeat && (remainder < bsize)) {
        memset(mSampleBfr1, 0x0, bsize);
    }

    n = fread (mSampleBfr1, 1, min(remainder, bsize), wd->file1);
    if (!repeat && n <= 0)
        return;
    if (repeat) {
        if (n < 0)
            return;
        if (n < bsize) {
            /* We play the file in a loop, so if the last chunk is less then the fragsize
             * then we zero pad the remainder of the fragment (set above memset) and
             * report n as the full fragsize
             */
            n = bsize;
        }
    }

    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));
    }
    /*
     * When written is smaller than n, we want to make sure we keep trying to write
     * so that we don't skip any data from the file. In blocking mode, we just
     * try a second time to write.
     * In nonblocking mode, we usleep frag_period_us / (WRITE_RETRY_TIMES -1),
     * then we just need to retry at most WRITE_RETRY_TIMES.
     */
    while (written < n && retries < WRITE_RETRY_TIMES)
    {
        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");
            cleanup_and_exit(EXIT_FAILURE);
        }

        switch (status.status)
        {
            case SND_PCM_STATUS_UNDERRUN:
            case SND_PCM_STATUS_READY:
                if( status.status == SND_PCM_STATUS_UNDERRUN ) {
                    printf ("Audio underrun occured\n");
                }
                if (snd_pcm_plugin_prepare (pcm_handle, SND_PCM_CHANNEL_PLAYBACK) < 0)
                {
                    fprintf (stderr, "underrun: playback channel prepare error\n");
                    cleanup_and_exit(EXIT_FAILURE);
                }
                break;
            case 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");
                    cleanup_and_exit(EXIT_FAILURE);
                }
                break;
            case SND_PCM_STATUS_ERROR:
                fprintf(stderr, "error: playback channel failure\n");
                cleanup_and_exit(EXIT_FAILURE);
                break;
            case SND_PCM_STATUS_PREEMPTED:
                fprintf(stderr, "error: playback channel preempted\n");
                cleanup_and_exit(EXIT_FAILURE);
                break;
            case SND_PCM_STATUS_CHANGE:
                printf ("Audio device change occured\n");
                if (snd_pcm_plugin_params (pcm_handle, &pp) < 0)
                {
                    fprintf (stderr, "device change: snd_pcm_plugin_params failed, why_failed = %d\n", pp.why_failed);
                    cleanup_and_exit(EXIT_FAILURE);
                }
                if (snd_pcm_plugin_prepare (pcm_handle, SND_PCM_CHANNEL_PLAYBACK) < 0)
                {
                    fprintf (stderr, "device change: playback channel prepare error\n");
                    cleanup_and_exit(EXIT_FAILURE);
                }
                break;
            case SND_PCM_STATUS_PAUSED:
                printf("Paused\n");
                /* Fall-Through - no break */
            case SND_PCM_STATUS_SUSPENDED:
                if (use_writer_thread)
                {
                   fd_set wfds;
                   int pcm_fd;

                   /* Wait until there is more room in the buffer (unsuspended/resumed) */
                   FD_ZERO(&wfds);
                   pcm_fd = snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_PLAYBACK);
                   FD_SET (pcm_fd, &wfds);
                   if (select (pcm_fd + 1, NULL, &wfds, NULL, NULL) == -1)
                       perror ("select");
                   continue; /* Go back to the top of the write loop */
                }
                else
                    return; /* Go back to higher level select() to wait until room in buffer (unsuspend/resume) */
                break;
             default:
                break;
        }

        if (written < 0)
            written = 0;
        if (nonblock)
            usleep(frag_period_us / (WRITE_RETRY_TIMES - 1));
        written += snd_pcm_plugin_write (pcm_handle, mSampleBfr1 + written, n - written);
        retries++;
    }
    N += written;
    if (repeat && N >= mDataSize) {
        /* We hit the end of file, rewind back to the beginning */
        fseek (wd->file1, data_position, SEEK_SET);
        N = 0;
    }
}

static void *writer_thread_handler(void *data)
{
    WriterData *wd = (WriterData *)data;
    sigset_t signals;

    sigfillset (&signals);
    pthread_sigmask (SIG_BLOCK, &signals, NULL);

    while (running && N < mDataSize && n > 0)
    {
        write_audio_data(wd);
    }

    return NULL;
}

static 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] wavfile

Options:
    -a[card#:]<dev#>   the card & device number to play out on
                       OR
    -a<name>           the symbolic name of the device 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
    -x                 use mmap interface
    -i                 Display PCM channel info
    -R<value>          SRC rate method
                       (1 = 7-pt kaiser windowed, 2 = 20-pt remez, 3 = linear interpolation)
    -y                 Nonblocking mode
    -o<audio type>     string name for Audio type
    -d                 Continuously repeat input file

Runtime Controls:
                       'p': Pause
                       'r': Resume
                       'i': Display status
                       't': Stop repeating input file

Volume Controls:
Adjust the volume by 10 or by the remaining volume steps until the min or max (whichever is smaller)
                       'q': increase volume on front left, rear left, woofer
                       'a': decrease volume on front left, rear left, woofer
                       'w': increase volume on front left, rear left, front right, rear right, front center, woofer
                       's': decrease volume on front left, rear left, front right, rear right, front center, woofer
                       'e': increase volume on front right, rear right, front center
                       'd': decrease volume on front right, rear right, front center

Voice Matrix Configuration 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* */
//*****************************************************************************

static void sig_handler( int sig_no )
{
    running = false;
    return;
}
```

```
static void dump_info( int card, int dev, const char* name, int channel )
{
    int rtn;
    snd_pcm_t * pcm_handle = 0;
    snd_pcm_channel_info_t pi;
    int open_mode = (channel == SND_PCM_CHANNEL_PLAYBACK)?SND_PCM_OPEN_PLAYBACK:SND_PCM_OPEN_CAPTURE;

    if (name[0] != '\0')
        rtn = snd_pcm_open_name(&pcm_handle, name, open_mode);
    else if (card == -1)
        rtn = snd_pcm_open_preferred(&pcm_handle, NULL, NULL, open_mode);
    else
        rtn = snd_pcm_open (&pcm_handle, card, dev, open_mode);
    if (rtn < 0)
    {
        fprintf(stderr, "Cannot open %s device - %s\n",
                (channel == SND_PCM_CHANNEL_PLAYBACK) ? "playback" : "capture", snd_strerror(rtn));
        return;
    }

    memset(&pi, 0, sizeof (pi));
    pi.channel = channel;
    if ((rtn = snd_pcm_channel_info( pcm_handle, &pi )) == 0)
    {
        snd_pcm_chmap_t *chmap;

        printf("\n%s %s Info\n", pi.subname, (channel == SND_PCM_CHANNEL_PLAYBACK) ? "Playback" : "Capture");
        printf("flags 0x%X\n", pi.flags);
        printf("formats 0x%X\n", pi.formats);
        printf("fragment_align %d\n", pi.fragment_align);
        printf("max_buffer_size %d\n", pi.max_buffer_size);
        printf("max_fragment_size %d\n", pi.max_fragment_size);
        printf("min_fragment_size %d\n", pi.min_fragment_size);
        printf("rates 0x%X\n", pi.rates);
        printf("max_rate %d\n", pi.max_rate);
        printf("min_rate %d\n", pi.min_rate);
        printf("max_voices %d\n", pi.max_voices);
        printf("min_voices %d\n", pi.min_voices);
        printf("subdevice %d\n", pi.subdevice);
        printf("transfer_block_size %d\n", pi.transfer_block_size);
        if (pi.mixer_gid.name && pi.mixer_gid.name[0] != '\0')
            printf("mixer_gid %s, %d\n", pi.mixer_gid.name, pi.mixer_gid.index);
        else
            printf("mixer_gid not set\n");

        chmap = snd_pcm_get_chmap( pcm_handle );
        if (chmap != NULL)
        {
            int i;
            printf("channel map:\n");
            for (i=0; i<chmap->channels; i++) {
                printf("%i\t%s\n", i, snd_pcm_get_chmap_channel_name(chmap->pos[i]));
            }
            free(chmap);
        }
    } else
        printf("Error retrieving %s info\n", (channel == SND_PCM_CHANNEL_PLAYBACK) ? "playback" : "capture");

    snd_pcm_close( pcm_handle );
}
```

```
int
main (int argc, char **argv)
{
    int     card = -1;
    int     dev = 0;
    FmtChunk fmt;
    int     mSampleRate;
    int     mSampleChannels;
    int     mSampleBits;
    int     mSampleBytes;
    int     fragsize = -1;
    int     fmtLength = 0;
    int     mode = SND_PCM_OPEN_PLAYBACK;

    int     rtn;
    snd_pcm_channel_info_t pi;
    snd_pcm_channel_setup_t setup;
    int c;
    fd_set  rfds, wfds, ofds;
#define MAX_VOICES 8
    uint32_t voice_mask[MAX_VOICES] = { 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;
    void   *retval;
    pthread_t writer_thread;
    pthread_t mixer_thread;
    pthread_t keypress_thread;
    pthread_t pcm_event_thread;
    char   type[sizeof(pp.audio_type_name)] = {0};
    int    rate_method = 0;
    int    use_mmap = 0;
    int    info = 0;
    struct stat fileStat;
    int    pcm_fd, mixer_fd = 0;
    snd_pcm_filter_t pevent;

    // Start logging
    memset(&slog_config, 0, sizeof(slog_config));
    slog_config.num_buffers = 1;
    slog_config.verbosity_level = SLOG2_INFO;
    slog_config.buffer_set_name = "wave";
    slog_config.buffer_config[0].buffer_name="wave";
    slog_config.buffer_config[0].num_pages = 5;
    slog2_register(&slog_config, &slog_handle, 0);
    slog2_set_verbosity(slog_handle, SLOG2_INFO);
    slog2_set_default_buffer(slog_handle);

    while ((c = getopt (argc, argv, "ia:ef:vc:n:b:p:m:rstwo:xR:yd")) != 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 'i':
            info = 1;
            break;
        case 'v':
            verbose = 1;
            break;
        case 'c':
            sub_opts = sub_opts_copy = strdup (optarg);
            if (sub_opts == NULL) {
                perror ("Cannot allocate sub_opts");
                return (EXIT_FAILURE);
            }
            while (*sub_opts != '\0')
            {
                int channel = getsubopt (&sub_opts, dev_opts, &value);
                if( (channel >= 0) && (channel < MAX_VOICES) && value ) {
                    voice_mask[channel] = strtoul (value, NULL, 0);
                } else {
                    fprintf (stderr, "Invalid channel map specified\n");
                    return (EXIT_FAILURE);
                }
            }
            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':
            strlcpy (mixer_name, optarg, sizeof(mixer_name));
            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':
            strlcpy (type, optarg, sizeof(type));
            type[sizeof(pp.audio_type_name) - 1] = '\0';
            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;
        case 'y':
            mode |= SND_PCM_OPEN_NONBLOCK;
            nonblock = 1;
            break;
        case 'd':
            repeat = 1;
            printf("Repeat playing\n");
            break;
        default:
            fprintf(stderr, "Invalid option -%c\n", c);
            return (EXIT_FAILURE);
        }
    }

    setvbuf (stdin, NULL, _IONBF, 0);
    if (info) {
        dump_info( card, dev, name, SND_PCM_CHANNEL_PLAYBACK);
        dump_info( card, dev, name, SND_PCM_CHANNEL_CAPTURE);
        return (EXIT_SUCCESS);
    }

    if (optind >= argc) {
        fprintf(stderr, "no file specified\n");
        return (EXIT_FAILURE);
    }

    if (name[0] != '\0')
    {
        snd_pcm_info_t info;

        if ((rtn = snd_pcm_open_name (&pcm_handle, name, mode)) < 0)
        {
            fprintf(stderr, "open_name failed - %s\n", snd_strerror(rtn));
            return (EXIT_FAILURE);
        }
        rtn = snd_pcm_info (pcm_handle, &info);
        card = info.card;
    }
    else
    {
        if (card == -1)
        {
            if ((rtn = snd_pcm_open_preferred (&pcm_handle, &card, &dev, mode)) < 0)
            {
                fprintf(stderr, "device open failed - %s\n", snd_strerror(rtn));
                return (EXIT_FAILURE);
            }
        }
        else
        {
            if ((rtn = snd_pcm_open (&pcm_handle, card, dev, mode)) < 0)
            {
                fprintf(stderr, "device open failed - %s\n", snd_strerror(rtn));
                return (EXIT_FAILURE);
            }
        }
    }

    if ((wd.file1 = fopen (argv[optind], "r")) == 0) {
        perror ("file open");
        cleanup_and_exit(EXIT_FAILURE);
    }

    if (stat (argv[optind], &fileStat) == -1) {
        perror ("file stat");
        cleanup_and_exit(EXIT_FAILURE);
    }

    if (CheckHdr (wd.file1) == -1) {
        fprintf (stderr, "invalid wav header\n");
        cleanup_and_exit(EXIT_FAILURE);
    }

    fmtLength = FindTag (wd.file1, "fmt ");
    if ((fmtLength == 0) || (fmtLength < sizeof(fmt)) ||
        (fread (&fmt, sizeof(fmt), 1, wd.file1) == 0)) {
```

```
        fprintf (stderr, "invalid wav file\n");
        cleanup_and_exit(EXIT_FAILURE);
    }

    /* Some files have extra data in fmt field, so skip past it */
    if (fmtLength > sizeof(fmt))
        fseek (wd.file1, fmtLength - sizeof(fmt), SEEK_CUR);

    mDataSize = FindTag (wd.file1, "data");
    data_position = ftell(wd.file1);
    if ((mDataSize == 0) || (mDataSize > ((long)fileStat.st_size - data_position))) {
        fprintf (stderr, "wave data size conflicts with file size mDataSize=%d file size=%zd, data_pos=%ld\n",
                 mDataSize, fileStat.st_size, data_position);
        cleanup_and_exit(EXIT_FAILURE);
    }

    if (mBigEndian) {
        mSampleRate = ENDIAN_BE32 (fmt.SamplesPerSec);
        mSampleChannels = ENDIAN_BE16 (fmt.Channels);
        mSampleBits = ENDIAN_BE16 (fmt.BitsPerSample);
        /* BlockAlign = frame size, i.e. sample size including padding bits, times number of channels */
        mSampleBytes = ENDIAN_BE16 (fmt.BlockAlign) / mSampleChannels;
        fmt.FormatTag = ENDIAN_BE16 (fmt.FormatTag);
    } else {
        mSampleRate = ENDIAN_LE32 (fmt.SamplesPerSec);
        mSampleChannels = ENDIAN_LE16 (fmt.Channels);
        mSampleBits = ENDIAN_LE16 (fmt.BitsPerSample);
        /* BlockAlign = frame size, i.e. sample size including padding bits, times number of channels */
        mSampleBytes = ENDIAN_LE16 (fmt.BlockAlign) / mSampleChannels;
        fmt.FormatTag = ENDIAN_LE16 (fmt.FormatTag);
    }

    printf("SampleRate = %d, Channels = %d, SampleBits = %d, SampleBytes = %d\n",
           mSampleRate, mSampleChannels, mSampleBits, mSampleBytes);
    printf("Playback Duration = %fs\n", (float)mDataSize / (mSampleRate * mSampleChannels * mSampleBytes));

    /* Enable PCM events */
    pevent.enable = SND_PCM_EVENT_MASK(SND_PCM_EVENT_AUDIOMGMT_STATUS) |
                    SND_PCM_EVENT_MASK(SND_PCM_EVENT_AUDIOMGMT_MUTE) |
                    SND_PCM_EVENT_MASK(SND_PCM_EVENT_OUTPUTCLASS) |
                    SND_PCM_EVENT_MASK(SND_PCM_EVENT_UNDERRUN);
    snd_pcm_set_filter(pcm_handle, SND_PCM_CHANNEL_PLAYBACK, &pevent);

    if (use_mmap)
    {
        snd_pcm_plugin_set_enable (pcm_handle, PLUGIN_MMAP);
    }

    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));
        cleanup_and_exit(EXIT_FAILURE);
    }

    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 (fmt.FormatTag == 6) {
        pp.format.format = SND_PCM_SFMT_A_LAW;
    } else if (fmt.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 && mSampleBytes == 4) {
        if (mBigEndian) {
            pp.format.format = SND_PCM_SFMT_S24_4_BE;
        } else {
            pp.format.format = SND_PCM_SFMT_S24_4_LE;
        }
    } else if (mSampleBits == 24 && mSampleBytes == 3) {
        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, sample size %d\n", mSampleBits, mSampleBytes);
        cleanup_and_exit(EXIT_FAILURE);
    }

    strlcpy (pp.audio_type_name, type, sizeof(pp.audio_type_name));

    if (mix_name_enable == 1)
    {
        strlcpy (pp.sw_mixer_subchn_name, mixer_name, sizeof(pp.sw_mixer_subchn_name));
    }
    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);
    cleanup_and_exit(EXIT_FAILURE);
}

if (voice_override)
{
    unsigned int i;
    snd_pcm_plugin_get_voice_conversion (pcm_handle, SND_PCM_CHANNEL_PLAYBACK,
        &voice_conversion);
    for (i = 0; i < MAX_VOICES; i++) {
        voice_conversion.matrix[i] = voice_mask[i];
    }
    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));
    cleanup_and_exit(EXIT_FAILURE);
}
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;
frag_period_us = (( (int64_t)bsize * 1000000 ) / (mSampleBytes * mSampleChannels * mSampleRate));
printf("Frag Period is %d us\n", frag_period_us);

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 (&mixer_handle, card, setup.mixer_device)) < 0)
    {
        fprintf (stderr, "snd_mixer_open failed: %s\n", snd_strerror (rtn));
        cleanup_and_exit(EXIT_FAILURE);
    }
}

if (tcgetpgrp (0) == getpid ()) {
    stdin_raw = 1;
```

```
        dev_raw (fileno (stdin));
    }

    if( print_timing ) {
        clock_gettime( CLOCK_REALTIME, &wd.start_time );
    }

    mSampleBfr1 = malloc (bsize);
    if ( mSampleBfr1 == NULL ) {
        perror("Failed to allocate pcm buffer");
        cleanup_and_exit(EXIT_FAILURE);
    }

    FD_ZERO (&rfds);
    FD_ZERO (&wfds);
    FD_ZERO (&ofds);
    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 ((rtn = snd_pcm_plugin_prepare (pcm_handle, SND_PCM_CHANNEL_PLAYBACK)) < 0)
    {
        fprintf (stderr, "snd_pcm_plugin_prepare failed: %s\n", snd_strerror (rtn));
        cleanup_and_exit(EXIT_FAILURE);
    }

    if( use_writer_thread ) {
        pthread_create( &writer_thread, NULL, writer_thread_handler, &wd );
        pthread_create( &pcm_event_thread, NULL, generic_thread_handler, handle_pcm_events );
        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);
        pthread_cancel(pcm_event_thread);
        if (mixer_handle)
            pthread_cancel(mixer_thread);
    } else {
        while (running && N < mDataSize && n > 0)
        {
            FD_ZERO(&rfds);
            FD_ZERO(&wfds);
            FD_ZERO(&ofds);
            if (stdin_raw)
                FD_SET (STDIN_FILENO, &rfds);
            if (mixer_handle) {
                mixer_fd = snd_mixer_file_descriptor (mixer_handle);
                FD_SET (mixer_fd, &rfds);
            }
            pcm_fd = snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_PLAYBACK);
            FD_SET (pcm_fd, &wfds);
            FD_SET (pcm_fd, &ofds);

            rtn = max (mixer_fd, pcm_fd);
            if (select (rtn + 1, &rfds, &wfds, &ofds, NULL) == -1)
            {
                perror ("select");
                break; /* break loop to exit cleanly */
            }

            if (FD_ISSET (STDIN_FILENO, &rfds))
            {
                handle_keypress();
            }

            if (mixer_handle && FD_ISSET (mixer_fd, &rfds))
            {
                handle_mixer();
            }

            if (FD_ISSET (pcm_fd, &wfds))
            {
                write_audio_data(&wd);
            }

            if (FD_ISSET (pcm_fd, &ofds))
            {
                snd_pcm_event_t event;
                if ((rtn = snd_pcm_channel_read_event (pcm_handle, SND_PCM_CHANNEL_PLAYBACK, &event)) == EOK)
                {
                    switch (event.type)
                    {
                        case SND_PCM_EVENT_AUDIOMGMT_STATUS:
                            display_status_event(&event);
                            break;
                        case SND_PCM_EVENT_AUDIOMGMT_MUTE:
                            display_mute_event(&event);
                            break;
                        case SND_PCM_EVENT_OUTPUTCLASS:
                            printf("Output class event received - output class changed from %d to %d\n",
                                   event.data.outputclass.old_output_class, event.data.outputclass.new_output_class);
                            break;
                        case SND_PCM_EVENT_UNDERRUN:
                            printf("Underrun event received\n");
                            break;
                        default:
                            printf("Unknown PCM event type - %d\n", event.type);
                            break;
                    }
                 }
                 else
                     printf("snd_pcm_channel_read_event() failed with %d\n", rtn);
            }
        }
    }

    printf("Exiting...\n");

    if (running)
        snd_pcm_plugin_flush (pcm_handle, SND_PCM_CHANNEL_PLAYBACK);
    cleanup();
    return(EXIT_SUCCESS);
}

#if defined(__QNXNTO__) && defined(__USESRCVERSION)
#include <sys/srcversion.h>
__SRCVERSION("$URL$ $Rev$")
#endif
```

Appendix D
waverec.c example

This is a sample application that captures (i.e., records) audio data.

For information about using this utility, see “*waverec*” in the *QNX Neutrino Utilities Guide*.

```
/*
 * $QNXLicenseC:
 * Copyright 2016, 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 <limits.h>
#include <ctype.h>

#include <sys/asoundlib.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* */

static snd_mixer_t *mixer_handle = NULL;
static snd_pcm_t *pcm_handle = NULL;
static snd_pcm_channel_params_t pp;
static char *mSampleBfr1 = NULL;
static FILE *file1 = NULL;
static int stdin_raw = 0;

static 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));
}

static 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));
}

static void
cleanup(void)
{
 if (stdin_raw)
  dev_unraw (fileno (stdin));
 if (mSampleBfr1)
  free(mSampleBfr1);
 if (mixer_handle)
  snd_mixer_close (mixer_handle);
 if (file1)
  fclose(file1);
 if (pcm_handle)
  snd_pcm_close (pcm_handle);
}

static void
cleanup_and_exit(int exit_code)
{
 cleanup();
 exit(exit_code);
}

//****************************************************************************
/* *INDENT-OFF* */
#ifdef __USAGE
%C [Options] wavfile

Options:
    -a[card#:]<dev#>   the card & device number to record from
                       OR
    -a[name]           the symbolic name of the device to record from
    -b <size>          Sample size (8, 16, 24, 32)
    -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
    -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
    -y                 Nonblocking mode

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
    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* */
//*****************************************************************************

volatile int end = 0;

static void sig_handler( int sig_no )
{
 end = 1;
 return;
}

static 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     i, j;
 int     card = -1;
 int     dev = 0;
 int     ret;

 unsigned int mSamples;
 int     mSampleRate;
 int     mSampleChannels;
 int     mSampleBits;
 int     mSampleBytes;
 int     mSampleTime;
 int     fragsize = -1;
 int     num_frags = -1;
 int     verbose = 0;
 int     mode = SND_PCM_OPEN_CAPTURE;

 int     rtn;
 int     mixer_fd = 0;
 int     pcm_fd;
 snd_pcm_channel_info_t pi;
 snd_mixer_group_t group;
 snd_pcm_channel_setup_t setup;
 int     bsize, N = 0, c;
#define MAX_VOICES 8
 uint32_t voice_mask[MAX_VOICES] = { 0 };
 struct {
  snd_pcm_chmap_t map;
  unsigned int pos[32];
 } map;
 snd_pcm_voice_conversion_t voice_conversion;
 int     voice_override = 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 };
 int interleave = 1;
 fd_set  rfds, ofds;
 int use_mmap = 0;
 int rate_method = 0;
 snd_pcm_filter_t pevent;

 mSampleRate = 44100;
 mSampleChannels = 2;
 mSampleBits = 16;
 mSampleBytes = 2;
 mSampleTime = 5;

 while ((c = getopt (argc, argv, "b:a:f:mn:r:t:vc:xi:R:z:y")) != EOF)
 {
  switch (c)
  {
  case 'b':
   mSampleBits = atoi (optarg);
   if (mSampleBits != 8 && mSampleBits != 16 && mSampleBits != 24 && mSampleBits != 32)
   {
    fprintf(stderr, "Invalid sample size, must be one of 8, 16, 24, 32\n");
    return (EXIT_FAILURE);
   }
   mSampleBytes = mSampleBits/8;
   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 = sub_opts_copy = strdup (optarg);
   if (sub_opts == NULL) {
    perror("Cannot allocate sub_opts");
    return (EXIT_FAILURE);
   }
   while (*sub_opts != '\0')
   {
    int channel = getsubopt (&sub_opts, dev_opts, &value);
    if( (channel >= 0) && (channel < MAX_VOICES) && value ) {
     voice_mask[channel] = strtoul (value, NULL, 0);
    } else {
     fprintf (stderr, "Invalid channel map specified\n");
     free(sub_opts_copy);
     return (EXIT_FAILURE);
    }
   }
   free(sub_opts_copy);
   voice_override = 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;
  case 'y':
   mode |= SND_PCM_OPEN_NONBLOCK;
   break;
  default:
   fprintf(stderr, "Invalid option -%c\n", c);
   return (EXIT_FAILURE);
  }
 }

 if (optind >= argc)
 {
  fprintf(stderr, "no file specified\n");
  return (EXIT_FAILURE);
 }

 if (name[0] != '\0')
 {
  snd_pcm_info_t info;

  if ((rtn = snd_pcm_open_name (&pcm_handle, name, mode)) < 0)
  {
   fprintf(stderr, "snd_pcm_open_name failed - %s\n", snd_strerror(rtn));
   return (EXIT_FAILURE);
  }
  rtn = snd_pcm_info (pcm_handle, &info);
  card = info.card;
 }
 else
 {
  if (card == -1)
  {
   if ((rtn = snd_pcm_open_preferred (&pcm_handle, &card, &dev, mode)) < 0)
   {
    fprintf(stderr, "snd_pcm_open_preferred failed - %s\n", snd_strerror(rtn));
    return (EXIT_FAILURE);
   }
  }
  else
  {
   if ((rtn = snd_pcm_open (&pcm_handle, card, dev, mode)) < 0)
   {
    fprintf(stderr, "snd_pcm_open failed - %s\n", snd_strerror(rtn));
    return (EXIT_FAILURE);
   }
  }
 }

 if ((file1 = fopen (argv[optind], "w")) == 0)
 {
```

```
 perror("file open failed");
 cleanup_and_exit(EXIT_FAILURE);
}

if( mSampleTime == 0 ) {
 mSamples = 0xFFFFFFFF - sizeof(riff_hdr) + 8;
} else {
 mSamples = mSampleRate * mSampleChannels * mSampleBytes * mSampleTime;
}

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 * mSampleBytes);
riff_hdr.wave.fmt.block_align = ENDIAN_LE16 (mSampleChannels * mSampleBytes);
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, SampleBytes = %d\n",
 mSampleRate, mSampleChannels, mSampleBits, mSampleBytes);

/* Enable PCM events */
pevent.enable = (1<<SND_PCM_EVENT_OVERRUN);
snd_pcm_set_filter(pcm_handle, SND_PCM_CHANNEL_CAPTURE, &pevent);

if (use_mmap)
{
 snd_pcm_plugin_set_enable (pcm_handle, PLUGIN_MMAP);
}

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));
 cleanup_and_exit(EXIT_FAILURE);
}

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));
 cleanup_and_exit(EXIT_FAILURE);
}

if (voice_override)
{
 snd_pcm_plugin_get_voice_conversion (pcm_handle, SND_PCM_CHANNEL_CAPTURE,
  &voice_conversion);
 for(i = 0; i < MAX_VOICES; i++) {
  voice_conversion.matrix[i] = voice_mask[i];
 }
 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));
 cleanup_and_exit(EXIT_FAILURE);
}
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;

map.map.channels = 32;
if((rtn = snd_pcm_query_channel_map(pcm_handle, &map.map)) == EOK) {
 printf("Channel map:");
 if((rtn = snd_pcm_plugin_get_voice_conversion (pcm_handle, SND_PCM_CHANNEL_CAPTURE, &voice_conversion)) != EOK) {
  // The hardware map is the same as the voice map
  for( i = 0; i < map.map.channels; i ++ ) {
   printf(" (%d)", map.map.pos[i]);
  }
  printf("\n");
 } else {
  // Map hardware channels according to the voice map
  for( i = 0; i < voice_conversion.app_voices; i ++ ) {
   bool printed = false;
   printf(" (");
   for( j = 0; j < voice_conversion.hw_voices; j ++ ) {
    if( voice_conversion.matrix[i] & (1<<j) ) {
     if ( printed ) {
      printf(" ");
     } else {
      printed = true;
     }
     printf("%d", map.map.pos[j]);
    }
   }
   printf(")");
  }
  printf("\n");
 }
}

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, setup.mixer_card, setup.mixer_device)) < 0)
 {
  fprintf (stderr, "snd_mixer_open failed: %s\n", snd_strerror (rtn));
  cleanup_and_exit(EXIT_FAILURE);
 }
}

if (tcgetpgrp (0) == getpid ()) {
 stdin_raw = 1;
 dev_raw (fileno (stdin));
}

mSampleBfr1 = malloc (bsize);
if ( mSampleBfr1 == NULL ) {
 perror("Failed to allocate pcm buffer");
 cleanup_and_exit(EXIT_FAILURE);
}

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));
 cleanup_and_exit(EXIT_FAILURE);
}

FD_ZERO (&rfds);
FD_ZERO(&ofds);
signal(SIGINT, sig_handler);
signal(SIGTERM, sig_handler);
while (!end && N < mSamples)
{
 /* If we are the foreground process group associated with STDIN then include
  * STDIN in the fdset to handle key presses.
  */
```

```
if (stdin_raw)
 FD_SET (STDIN_FILENO, &rfds);
if (mixer_handle)
{
 /* Include the mixer_handle descriptor in the fdset to handle
  * mixer events.
  */
 mixer_fd = snd_mixer_file_descriptor (mixer_handle);
 FD_SET (mixer_fd, &rfds);
}
rtn = pcm_fd = snd_pcm_file_descriptor (pcm_handle, SND_PCM_CHANNEL_CAPTURE);
FD_SET (pcm_fd, &rfds);
FD_SET (pcm_fd, &ofds);

if (mixer_handle)
 rtn = max (mixer_fd, pcm_fd);

if (select (rtn + 1, &rfds, NULL, &ofds, NULL) == -1)
{
 perror("select");
 break; /* break loop to exit cleanly */
}

if (FD_ISSET (STDIN_FILENO, &rfds))
{
 c = getc (stdin);
 if (c != EOF)
 {
  /* Only handle volume key presses if there is a mixer group */
  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;
   default:
    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));
    }

    switch (c)
    {
    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;
    default:
     break;
    }
```

```
      }
      else {
       cleanup_and_exit(EXIT_SUCCESS);
      }
     }

     if (mixer_handle && FD_ISSET (mixer_fd, &rfds))
     {
      snd_mixer_callbacks_t callbacks = {
       0, 0, 0, 0
      };

      snd_mixer_read (mixer_handle, &callbacks);
     }

     if (FD_ISSET (pcm_fd, &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");
        cleanup_and_exit(EXIT_FAILURE);
       }
       if (status.status == SND_PCM_STATUS_CHANGE ||
        status.status == SND_PCM_STATUS_READY ||
        status.status == SND_PCM_STATUS_OVERRUN) {
        if (status.status == SND_PCM_STATUS_CHANGE) {
         fprintf(stderr, "change: capture channel capability change\n");
         if (snd_pcm_plugin_params (pcm_handle, &pp) < 0)
         {
          fprintf (stderr, "Capture channel snd_pcm_plugin_params error\n");
          cleanup_and_exit(EXIT_FAILURE);
         }
        }
        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");
         cleanup_and_exit(EXIT_FAILURE);
        }
       }
       else if (status.status == SND_PCM_STATUS_ERROR)
       {
        fprintf(stderr, "error: capture channel failure\n");
        cleanup_and_exit(EXIT_FAILURE);
       }
       else if (status.status == SND_PCM_STATUS_PREEMPTED)
       {
        fprintf(stderr, "error: capture channel preempted\n");
        cleanup_and_exit(EXIT_FAILURE);
```

```
    }
   } else {
    fwrite (mSampleBfr1, 1, read, file1);
    N += read;
   }
  }
  if (FD_ISSET (pcm_fd, &ofds))
  {
   snd_pcm_event_t event;
   if ((rtn = snd_pcm_channel_read_event (pcm_handle, SND_PCM_CHANNEL_CAPTURE, &event)) == EOK)
   {
    switch (event.type)
    {
     case SND_PCM_EVENT_OVERRUN:
      printf("Overrun event received\n");
      break;
     default:
      printf("Unknown PCM event type for capture - %d\n", event.type);
      break;
    }
   }
   else
    printf("snd_pcm_channel_read_event() failed with %d\n", rtn);
  }
 }

 printf("Exiting...\n");

 /* Update wave header with actual length of audio captured */
 riff_hdr.wave.data.data_len = ENDIAN_LE32 (N);
 riff_hdr.wave_len = ENDIAN_LE32 (N + sizeof (riff_hdr) - 8);
 fseek(file1, 0, SEEK_SET);
 fwrite (&riff_hdr, 1, sizeof (riff_hdr), file1);

 snd_pcm_plugin_flush (pcm_handle, SND_PCM_CHANNEL_CAPTURE);
 cleanup();
 return(EXIT_SUCCESS);
}

#if defined(__QNXNTO__) && defined(__USESRCVERSION)
#include <sys/srcversion.h>
__SRCVERSION("$URL$ $Rev$")
#endif
```

Appendix E
mix_ctl.c example

This is a sample application that captures the groups and switches in the mixer.

For information about using this utility, see `mix_ctl` in the *QNX Neutrino Utilities Reference*.

```
/*
 * $QNXLicenseC:
 * Copyright 2016, 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 <ctype.h>

#include <sys/asoundlib.h>


//*****************************************************************************
/* *INDENT-OFF* */
#ifdef __USAGE
%C  [Options] Cmds

Options:
    -a[card#:]<dev#>  the card & mixer device number to access
                      OR
    -a[name]          the AFM card name (e.g. voice, icc) 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 voice number Y (zero-based) is optional and
          restricts the change to the specified voice Y (if possible)

    switches

    switch  name  [value]
        - name is the switch name quoted if it contains white space
#endif
/* *INDENT-ON* */
//*****************************************************************************


static void
display_group (snd_mixer_t * mixer_handle, snd_mixer_gid_t * gid, snd_mixer_group_t * group)
{
 int j;
 int dB_scale_factor = 100;
 double percent, dB;

 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_JOINTLY_VOLUME)
  printf (" Jointly-Volume");
 else 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_BALANCE)
  printf (" Balance");
 if (group->caps & SND_MIXER_GRPCAP_FADE)
  printf (" Fade");
 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 - ");
 if (group->channels)
 {
  int chn_cnt = 0;
  for (j = 0; j <= SND_MIXER_CHN_LAST; j++)
  {
   if (!(group->channels & (1 << j)))
    continue;
   if ( (chn_cnt!=0) && ((chn_cnt%8) == 0) ) /* Display 8 channel names per line */
    printf("\n\t\t   ");
   printf ("%s ", snd_mixer_channel_name (j));
   chn_cnt++;
  }
 }
 else
 {
  printf("None");
 }
 printf ("\n");

 if (group->dB_scale_factor > 0) dB_scale_factor = group->dB_scale_factor;
 printf ("\tVolume Range - minimum=%i (%.3fdB), maximum=%i (%.3fdB)\n",
   group->min, (double)group->min_dB/dB_scale_factor, group->max, (double)group->max_dB/dB_scale_factor);

 for (j = 0; j <= SND_MIXER_CHN_LAST; j++)
 {
  if (!(group->channels & (1 << j)))
   continue;
  percent = (group->max - group->min) <= 0 ? 0.0 : 100.0 * (double)(group->volume.values[j] - group->min)
    / (double)(group->max - group->min);
  dB = (((percent / 100.0) * (double)(group->max_dB - group->min_dB)) + (double)group->min_dB) / dB_scale_factor;
  printf ("\tChannel %2d %-22.22s - %3d (%3d%%) (%.3fdB) %s %s\n", j,
   snd_mixer_channel_name (j), group->volume.values[j],
   (int)percent, dB,
   group->mute & (1 << j) ? "Muted" : "", group->capture & (1 << j) ? "Capture" : "");
 }
 if (group->caps & (SND_MIXER_GRPCAP_BALANCE|SND_MIXER_GRPCAP_FADE))
 {
  printf("\tBalance/Fade Range - minimum=%i, maximum=%i\n", group->balance_min, group->balance_max);
  if (group->caps & SND_MIXER_GRPCAP_BALANCE)
  {
   percent = (group->balance_max - group->balance_min) <= 0 ? 0.0 : 100.0 * (double)(group->balance_level - group->balance_min)
     / (double)(group->balance_max - group->balance_min);
   printf("\t\tBalance Level - %3d (%3d%%)\n", group->balance_level, (int)percent);
  }
  if (group->caps & SND_MIXER_GRPCAP_FADE)
  {
   percent = (group->balance_max - group->balance_min) <= 0 ? 0.0 : 100.0 * (double)(group->fade_level - group->balance_min)
     / (double)(group->balance_max - group->balance_min);
   printf("\t\tFade Level    - %3d (%3d%%)\n", group->fade_level, (int)percent);
  }
 }
}


static 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 ((rtn = snd_mixer_groups (mixer_handle, &groups)) < 0)
   {
    fprintf (stderr, "snd_mixer_groups failed: %s\n", snd_strerror (rtn));
   }
   else if (groups.groups == 0)
   {
    fprintf (stderr, "--> No mixer groups to list <-- \n");
    break;
   }

   if (groups.groups_over > 0)
   {
    groups.pgroups =
     (snd_mixer_gid_t *) malloc (sizeof (snd_mixer_gid_t) * groups.groups);
    if (groups.pgroups == NULL)
    {
     fprintf (stderr, "Unable to malloc group array - %s\n", strerror (errno));
     groups.groups = 0;
     break;
    }
    groups.groups_size = groups.groups;
    groups.groups_over = 0;
    groups.groups = 0;
    if ((rtn = snd_mixer_groups (mixer_handle, &groups)) < 0)
    {
     fprintf (stderr, "snd_mixer_groups failed: %s\n", snd_strerror (rtn));
     groups.groups = 0;
     break;
    }
    if (groups.groups_over > 0) /* Mixer controls have changed since call above, try again */
    {
     free (groups.pgroups);
     continue;
    }
    snd_mixer_sort_gid_table (groups.pgroups, groups.groups, 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 of group %d failed: %s\n", i, snd_strerror (rtn));
     continue;
    }

    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,
      (int)(2 + sizeof (groups.pgroups[i].name) - strlen (groups.pgroups[i].name)), ' ',
      group.caps & SND_MIXER_GRPCAP_PLAY_GRP ? "Playback Group" : "Capture Group");
    }
   }
  }
 }


 static 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;
  int     rtn;

  while (1)
  {
   memset (&groups, 0, sizeof (groups));
   if ((rtn = snd_mixer_groups (mixer_handle, &groups)) < 0)
   {
    fprintf (stderr, "snd_mixer_groups failed: %s\n", snd_strerror (rtn));
   }
   else if (groups.groups == 0)
   {
    break;
   }

   if (groups.groups_over > 0)
   {
    groups.pgroups =
     (snd_mixer_gid_t *) malloc (sizeof (snd_mixer_gid_t) * groups.groups);
    if (groups.pgroups == NULL)
    {
     fprintf (stderr, "Unable to malloc group array - %s\n", strerror (errno));
     groups.groups = 0;
     break;
    }
    groups.groups_size = groups.groups;
    groups.groups_over = 0;
    groups.groups = 0;
    if ((rtn = snd_mixer_groups (mixer_handle, &groups)) < 0)
    {
     fprintf (stderr, "snd_mixer_groups failed: %s\n", snd_strerror (rtn));
     groups.groups = 0;
     break;
    }
    if (groups.groups_over > 0) /* Mixer controls have changed since call above, try again */
    {
     free (groups.pgroups);
     continue;
    }
    break;
   }
  }

  for (i = 0; i < groups.groups; i++)
  {
   if (strcasecmp (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;
 }


 static int
 group_option_value (char *option)
 {
  char   *ptr;
  int     value;

  if ((ptr = strrchr (option, '=')) != NULL)
  {
   if (*(ptr + 1) == 0)
    value = -2;
   else if (strcasecmp (ptr + 1, "off") == 0)
    value = 0;
   else if (strcasecmp (ptr + 1, "on") == 0)
    value = 1;
   else
    value = atoi (ptr + 1);
  }
  else
   value = -1;
  return (value);
 }

 static 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;
  long    channel = 0, j;
  int     value;
```

```
char    modified = 0;

if (optind >= argc)
{
 fprintf (stderr, "No Group specified \n");
 return;
}

memset (&gid, 0, sizeof (gid));
ptr = strtok (argv[optind++], ",");
if (ptr != NULL) {
 strlcpy (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;
 }
}

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 (strncasecmp (argv[optind], "mute", 4) == 0)
 {
  if (argv[optind][4] == '=')
   channel = LONG_MAX;
  else
  {
   channel = atoi (&argv[optind][4]);

   if (group.caps & SND_MIXER_GRPCAP_JOINTLY_MUTE)
    channel = LONG_MAX;
  }
  if (channel == LONG_MAX)
   group.mute = value ? group.channels : 0;
  else if (group.channels & (1<<channel))
  {
   group.mute = value ? group.mute | (1 << channel) : group.mute & ~(1 << channel);
  }
 }
 else if (strncasecmp (argv[optind], "capture", 7) == 0)
 {
  if (argv[optind][7] == '=')
   channel = LONG_MAX;
  else
  {
   channel = atoi (&argv[optind][7]);
```

```
    if (group.caps & SND_MIXER_GRPCAP_JOINTLY_CAPTURE)
     channel = LONG_MAX;
   }
   if (channel == LONG_MAX)
    group.capture = value ? group.channels : 0;
   else if (group.channels & (1<<channel))
   {
    group.capture =
     value ? group.capture | (1 << channel) : group.capture & ~(1 << channel);
   }
  }
  else if (strncasecmp (argv[optind], "volume", 6) == 0)
  {
   if (argv[optind][6] == '=')
    channel = LONG_MAX;
   else {
    channel = atoi (&argv[optind][6]);
    if ((group.caps & SND_MIXER_GRPCAP_JOINTLY_VOLUME) && (group.channels & (1<<channel)))
     channel = LONG_MAX;
   }
   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 ((strncasecmp (argv[optind], "balance", 7) == 0) && (group.caps & SND_MIXER_GRPCAP_BALANCE))
  {
   if (argv[optind][strlen (argv[optind]) - 1] == '%' && (group.balance_max - group.balance_min) >= 0)
    value = (value * (group.balance_max - group.balance_min)) / 100 + group.balance_min;
   if (value > group.balance_max)
    value = group.balance_max;
   if (value < group.balance_min)
    value = group.balance_min;
   group.balance_level = value;
  }
  else if ((strncasecmp (argv[optind], "fade", 4) == 0) && (group.caps & SND_MIXER_GRPCAP_FADE))
  {
   if (argv[optind][strlen (argv[optind]) - 1] == '%' && (group.balance_max - group.balance_min) >= 0)
    value = (value * (group.balance_max - group.balance_min)) / 100 + group.balance_min;
   if (value > group.balance_max)
    value = group.balance_max;
   if (value < group.balance_min)
    value = group.balance_min;
   group.fade_level = value;
  }
  else if (strncasecmp (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 [%ld] Not in group <<<<\n\n", channel);
  optind++;
 }

 /* 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 */
 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);
}


static void
display_switch (snd_switch_t * sw, char table_formated)
{
 printf ("\"%s\"%*c ", sw->name,
  (int)(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:
  {
   int i;
   if (sw->subtype == SND_SW_SUBTYPE_HEXA) {
    printf ("%s  0x%x ", "LIST   ", sw->value.list.data);
    for (i=0; i < sw->value.list.items_cnt; i ++) {
     printf(" 0x%x", sw->value.list.items[i]);
    }
   } else {
    printf ("%s  %d ", "LIST   ", sw->value.list.data);
    for (i=0; i < sw->value.list.items_cnt; i ++) {
     printf(" %d", sw->value.list.items[i]);
    }
   }
   printf("\n");
  }
  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);
  }
}


static void
display_switches (snd_ctl_t * ctl_handle, int mixer_dev, int argc, char *argv[])
{
 int     i;
 snd_switch_list_t list;
 snd_switch_t sw;
 int     rtn;

 while (1)
 {
  memset (&list, 0, sizeof (list));
  if ((rtn = snd_ctl_mixer_switch_list (ctl_handle, mixer_dev, &list)) < 0)
  {
   fprintf (stderr, "snd_ctl_mixer_switch_list failed: %s\n", snd_strerror (rtn));
  }
  else if (list.switches == 0)
  {
   fprintf (stderr, "--> No mixer switches to list <-- \n");
   break;
  }

  if (list.switches_over > 0)
  {
   list.pswitches = malloc (sizeof (snd_switch_list_item_t) * list.switches);
   if (list.pswitches == NULL)
   {
    fprintf (stderr, "Unable to malloc switch array - %s\n", strerror (errno));
    list.switches = 0;
    break;
   }
   list.switches_size = list.switches;
   list.switches_over = 0;
   list.switches = 0;
   if ((rtn = snd_ctl_mixer_switch_list (ctl_handle, mixer_dev, &list)) < 0)
   {
    fprintf (stderr, "snd_ctl_mixer_switch_list failed: %s\n", snd_strerror (rtn));
    list.switches = 0;
    break;
   }
   if (list.switches_over > 0) /* Mixer controls have changed since call above, try again */
   {
    free (list.pswitches);
    continue;
   }
   break;
  }
 }

 for (i = 0; i < list.switches; i++)
 {
  memset (&sw, 0, sizeof (sw));
  strlcpy (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 of switch %d failed: %s\n", i, snd_strerror (rtn));
   continue;
  }
  display_switch (&sw, 1);
 }
}


static 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 specified \n");
  return;
 }

 memset (&sw, 0, sizeof (sw));
 strlcpy (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 (strcasecmp (argv[optind], "off") == 0)
   value = 0;
  else if (strcasecmp (argv[optind], "on") == 0)
   value = 1;
  else if (strncasecmp (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) {
   int i;
   sw.value.list.data = value;
   for (i = optind; i < argc; i ++ ) {
    sw.value.list.items[i-optind] = atoi(argv[i]);
   }
   sw.value.list.items_cnt = argc - optind;
  }
  else if (sw.type == SND_SW_TYPE_STRING_11)
  {
   if (!string)
   {
    fprintf (stderr, "string required for switch type");
   }
   else
   {
    for (rtn = 0; rtn < sw.value.string_11.strings_cnt; rtn++)
    {
     if (strcasecmp (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;
 snd_mixer_info_t info = {0};
 char    name[_POSIX_PATH_MAX] = { 0 };

 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 if (isalpha (optarg[0]))
```

```
    strlcpy (name, optarg, sizeof(name));
   else
    dev = atoi (optarg);
   break;
  default:
   return 1;
  }
 }

 if (name[0] != '\0')
 {
  card = snd_card_name( name );
 }

 if ((rtn = snd_ctl_open (&ctl_handle, card)) < 0)
 {
  fprintf (stderr, "snd_ctl_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;
 }

 snd_mixer_info(mixer_handle, &info);
 if (name[0] != '\0')
  printf ("Using card %s (%d:%d), Mixer %s\n", name, card, dev, info.name);
 else
  printf ("Using card %d:%d, Mixer %s\n", card, dev, info.name);

 if (optind >= argc)
  display_groups (mixer_handle, argc - optind, argv + optind);
 else if (strcasecmp (argv[optind], "groups") == 0)
  display_groups (mixer_handle, argc - optind, argv + optind);
 else if (strcasecmp (argv[optind], "group") == 0)
  modify_group (mixer_handle, argc - optind, argv + optind);
 else if (strcasecmp (argv[optind], "switches") == 0)
  display_switches (ctl_handle, dev, argc - optind, argv + optind);
 else if (strcasecmp (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);
}

#if defined(__QNXNTO__) && defined(__USESRCVERSION)
#include <sys/srcversion.h>
__SRCVERSION("$URL$ $Rev$")
#endif
```

Appendix F
audiomgmt_monitor.c example

This is a sample application shows how to get the current active audio types.

You can use the following options to run the program:

- -a<*cardno*>, which specifies the card number for any control device belonging to the instance of io-audio that the ducking output belongs to.
- -n<*output_id*>, which specifies the ducking output ID to monitor

```
/*
 * $QNXLicenseC:
 * Copyright 2017, QNX Software Systems.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You
 * may not reproduce, modify or distribute this software except in
 * compliance with the License. You may obtain a copy of the License
 * at: http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" basis,
 * WITHOUT WARRANTIES OF ANY KIND, either express or implied.
 *
 * This file may contain contributions from others, either as
 * contributors under the License or as licensors under other terms.
 * 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 <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>
#include <errno.h>
#include <sys/asoundlib.h>

//*****************************************************************************
/* *INDENT-OFF* */
#ifdef __USAGE
%C [Options]

Options:
   -a<cardno>     Card number
   -n<output_id>  Ducking output ID to monitor
   -l             Use legacy status events

#endif
/* *INDENT-ON* */
//*****************************************************************************

static void audiomgmt_cb ( snd_ctl_t *hdl, void *private_data, int cmd)
{
    int status, i, j;
    const char *ducking_output = private_data;

    switch (cmd)
    {
        case SND_CTL_READ_AUDIOMGMT_CHG:    /* deprecated  - Use SND_CTL_READ_AUDIOMGMT_STATUS_CHG */
        {
            snd_ducking_status_t *info = NULL;
            snd_ducking_type_status_t *active_type = NULL;

            if ((status = snd_ctl_ducking_read(hdl, ducking_output, &info )) != EOK)
            {
                printf("Failed to read ducking info - %s\n", snd_strerror(status));
                return;
            }

            printf("\nActive audio types = %d\n", info->ntypes);
            /* The active_types structure is variable in size based on the number of pids it references, so we must
             * walk the active_type member of the info structure using the SND_DUCKING_STATUS_NEXT_TYPE() marco
             */
            for (i = 0, active_type = info->active_types; i < info->ntypes; i++,
                 active_type = SND_DUCKING_STATUS_NEXT_TYPE(active_type))
            {
                printf("Audio Type %s, Priority %d\n", active_type->name, active_type->prio);
                printf("\tPids (%d): ", active_type->npids);
                for (j = 0; j < active_type->npids; j++)
                {
                    printf("%d ", active_type->pids[j]);
                }
                printf("\n");
            }
            /* snd_ctl_ducking_read() will allocate the info buffer, so we must free it */
            free(info);
            break;
        }
        case SND_CTL_READ_AUDIOMGMT_STATUS_CHG:
        {
            snd_ducking_priority_status_t *info = NULL;

            if ((status = snd_ctl_ducking_status_read(hdl, ducking_output, &info )) != EOK)
            {
                printf("Failed to read ducking info - %s\n", snd_strerror(status));
                return;
            }

            printf("Active Subchns (%d):\n", info->nsubchns);
            /* The priority status structure is variable in size based on the number of subchns it references, so we must
             * walk the entries member of the info structure using the SND_DUCKING_STATUS_NEXT_ENTRY() marco
             */
            for (i = 0; i < info->nsubchns; i++)
            {
                printf("\tPriority: %d\n", info->subchns[i].prio);
                printf("\tPid: %d\n", info->subchns[i].pid);
                printf("\tAudio Type: %s\n", info->subchns[i].name);
                printf("\tState: ");
                if (info->subchns[i].state & SND_PCM_DUCKING_STATE_FORCED_ACTIVE)
                    printf("FORCED_ACTIVE ");
                else
                    printf("ACTIVE ");
                if (info->subchns[i].state & SND_PCM_DUCKING_STATE_DUCKED)
                    printf("| DUCKED ");
                if (info->subchns[i].state & SND_PCM_DUCKING_STATE_HARD_SUSPENDED)
                    printf("| HARD_SUSPENDED ");
                if (info->subchns[i].state & SND_PCM_DUCKING_STATE_SOFT_SUSPENDED)
                    printf("| SOFT_SUSPENDED ");
                if (info->subchns[i].state & SND_PCM_DUCKING_STATE_PAUSED)
                    printf("| PAUSED ");
                if (info->subchns[i].state & (SND_PCM_DUCKING_STATE_MUTE_BY_HIGHER|SND_PCM_DUCKING_STATE_MUTE_BY_SAME))
                    printf("| MUTED ");

                if (info->subchns[i].ducked_by & (SND_PCM_DUCKED_BY_SAME_PRIO|SND_PCM_DUCKED_BY_HIGHER_PRIO))
                {
                    printf("\n\tDucked by: ");
                    if (info->subchns[i].ducked_by & SND_PCM_DUCKED_BY_SAME_PRIO)
                        printf("SAME PRIORITY ");
                    else if (info->subchns[i].ducked_by & SND_PCM_DUCKED_BY_HIGHER_PRIO)
                        printf("HIGHER PRIORITY ");
                    if (info->subchns[i].ducked_by & SND_PCM_DUCKED_BY_NONTRANSIENT)
                        printf("| NON-TRANSIENT");
                    else
                        printf("| TRANSIENT");
                }
                printf("\n\n");
            }
            printf("\n");

            /* snd_ctl_ducking_status_read() will allocate the info buffer, so we must free it */
            free(info);
            break;
        }
        default:
            break;
    }
}

int main (int argc, char *argv[])
{
    int c;
    int legacy = 0;
    fd_set rfds;
    int card = 0, ctl_fd;
    snd_ctl_t *handle;
    snd_ctl_callbacks_t callbacks;
    snd_ctl_filter_t filter;
    char *ducking_output = NULL;

    optind = 1;
    while ((c = getopt(argc, argv, "a:n:l")) != EOF)
    {
        switch (c)
        {
            case 'a':
                card = strtol (optarg, &optarg, 0);
                break;
            case 'n':
                ducking_output = optarg;
                break;
            case 'l':
                legacy = 1;
                break;
            default:
                break;
        }
    }
    if (ducking_output == NULL)
    {
        printf("Ducking output not provided\n");
        return 1;
    }
    if (snd_ctl_open(&handle, card) != EOK)
    {
        printf("Failed to open control device on card %d\n", card);
        return 1;
    }
    ctl_fd = snd_ctl_file_descriptor(handle);
    memset(&callbacks, 0, sizeof (callbacks));
    callbacks.private_data = ducking_output;
    callbacks.audiomgmt = &audiomgmt_cb;

    /* Enable audio management status events */
    if (legacy)
        filter.enable = SND_CTL_EVENT_MASK(SND_CTL_READ_AUDIOMGMT_CHG);
    else
        filter.enable = SND_CTL_EVENT_MASK(SND_CTL_READ_AUDIOMGMT_STATUS_CHG);
    snd_ctl_set_filter(handle, &filter);

    FD_ZERO(&rfds);
    while (1)
    {
        FD_SET(ctl_fd, &rfds);
        if (select(ctl_fd + 1, &rfds, NULL, NULL, NULL) == -1)
        {
            printf("Select failed - %s\n", strerror(errno));
            return 1;
        }

        if (FD_ISSET(ctl_fd, &rfds))
        {
            snd_ctl_read(handle, &callbacks);
        }
    }
    snd_ctl_close (handle);
    return 0;
}

#if defined(__QNXNTO__) && defined(__USESRCVERSION)
#include <sys/srcversion.h>
__SRCVERSION("$URL$ $Rev$")
#endif
```

Appendix G
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 H
Glossary

ACS

Acoustics Control Server, a link between LiveAMP and the AFM modules in `io-audio`.

ADC

Analog Digital Converter. This converts an analog audio signal into a digital stream of samples.

AFM

AMP Functional Module.

ALSA

Advanced Linux Sound Architecture.

AMP

Acoustics Management Platform.

APX

Acoustic Processing Extension.

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