QNX® Momentics® DDK

Audio Devices

For targets running QNX® Neutrino® 6.3.0 or later

Printed under license by:

QNX Software Systems Co.
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: **info@qnx.com**
Web: **http://www.qnx.com/**

Publishing history

Electronic edition published 2005.

Technical support options

If you have any questions, comments, or problems with a QNX product, please contact Technical Support. For more information, see the How to Get Help chapter of the *Welcome to QNX Momentics* guide or visit our website, **www.qnx.com**.

Contents

List of Figures

About the Audio DDK

The following table may help you find information quickly:

If you want to:	Go to:
Estimate how much time it will take to develop an audio driver for your card	Evaluating Your Card
Understand how the audio driver fits into the QNX architecture	Organization of a driver
Write an analog mixer	Handling Analog Audio Data
Use a standard mixer DLL	Handling Analog Audio Data
Code a Pulse Code Modulation (PCM) device	Handling PCM Audio Data
Get details about the API functions	API Reference
Read about the codecs supported by QNX	Supported Codecs
Explore the code of a sample mixer	Sample Mixer Source
Understand the terms used in this guide	Glossary

For information about other audio drivers, see "Audio drivers (**`deva-*`**)" in the summary chapter of the QNX *Utilities Reference*.

Supported features

Currently supported features include:

- digital audio (PCM) playback and capture
- analog mixing controls
- routing of Digital Audio Data (AC3) through S/PDIF.

This DDK doesn't support:

- MIDI
- 3D audio
- game ports

Assumptions

To use this guide, you need to have:

- a basic familiarity with audio cards, terminology (e.g. DMA transfer, codec, analog signal, and interrupts), and basic audio operations
- sufficient hardware documentation for your audio chip in order to be able to program all the registers
- a working knowledge of the C programming language.

You might also need to refer to the QNX *Audio Developer's Guide*.

Building DDKs

You can compile the DDK from the IDE or the command line.

- To compile the DDK from the IDE:

  Please refer to the Managing Source Code chapter, and "QNX Source Package" in the Common Wizards Reference chapter of the *IDE User's Guide*.

- To compile the DDK from the command line:

  Please refer to the release notes or the installation notes for information on the location of the DDK archives.

  DDKs are simple zipped archives, with no special requirements. You must manually expand their directory structure from the

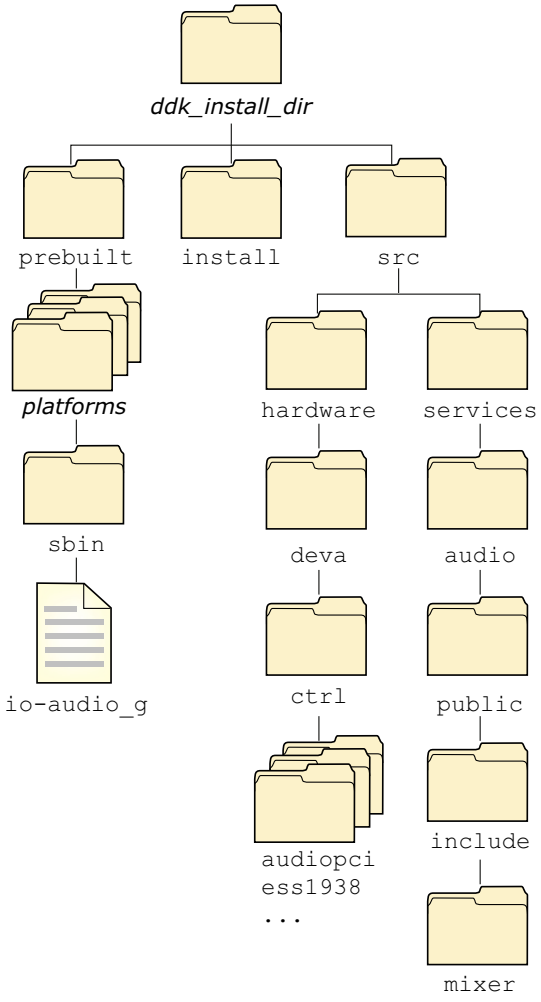
archive. You can install them into whichever directory you choose, assuming you have write permissions for the chosen directory.

Historically, DDKs were placed in **/usr/src/ddk_VERSION** directory, e.g. **/usr/src/ddk-6.2.1.** This method is no longer required, as each DDK archive is completely self-contained.

The following example indicates how you create a directory and unzip the archive file:

```
# cd ~
# mkdir my_DDK
# cd my_DDK
# unzip /path_to_ddks/ddk-device_type.zip
```

The top-level directory structure for the DDK looks like this:



Directory structure for this DDK.

You must run:

```
. ./setenv.sh
```

before running **make**, or **make install**.

Additionally, on Windows hosts you'll need to run the **Bash** shell (**bash.exe**) before you run the **. ./setenv.sh** command.

If you fail to run the **. ./setenv.sh** shell script prior to building the DDK, you can overwrite existing binaries or libs that are installed in **$QNX_TARGET**.

Each time you start a new shell, run the **. ./setenv.sh** command. The shell needs to be initialized before you can compile the archive.

The script will be located in the same directory where you unzipped the archive file. It must be run in such a way that it modifies the current shell's environment, not a sub-shell environment.

In **ksh** and **bash** shells, All shell scripts are executed in a sub-shell by default. Therefore, it's important that you use the syntax

```
. <script>
```

which will prevent a sub-shell from being used.

Each DDK is rooted in whatever directory you copy it to. If you type **make** within this directory, you'll generate all of the buildable entities within that DDK no matter where you move the directory.

all binaries are placed in a scratch area within the DDK directory that mimics the layout of a target system.

When you build a DDK, everything it needs, aside from standard system headers, is pulled in from within its own directory. Nothing that's built is installed outside of the DDK's directory. The makefiles shipped with the DDKs copy the contents of the **prebuilt** directory into the **install** directory. The binaries are built from the source using include files and link libraries in the **install** directory.

Chapter 1

Evaluating Your Card

The process of evaluating will help you estimate the amount of effort required to produce a QNX driver:

- Is your card on the PCI bus?

  If yes, the effort required is reduced because the PCI is a well standardized bus, and the card probably has on-chip DMA. Use the standard template driver as your starting point.

  If your card is on the ISA bus, the effort required is increased because you may need to worry about ISA PnP issues or other configuration issues to find and access your chip. Additionally, the chip probably requires the CPU DMA controller to transfer data. Use the Sound Blaster (**sb**) driver as your starting point.

  If your card is neither PCI or ISA, there's no specific template targeted at your situation. You'll have to learn from both the PCI template and Sound Blaster driver and adapt the knowledge to your specific situation.

- Does your card use a standard Codec (e.g. AC97 or AK4531)?

  If yes, your work will be easier because the DDK provides DLLs to control these codecs. If your card is PCI-based, use the standard template driver as your starting point. If the card isn't PCI-based, you have to create a hybrid from both templates. For more information, see the Supported Codecs appendix.

  If no, you'll have to write all the mixer control for this codec yourself. Use the Sound Blaster driver as your starting point. If the card is PCI-based, you'll have to create a hybrid from both templates.

- Does your card support playback of only one stream?

  Handling a multistream or subchannel card requires a little more effort and planning but provides the most economical driver. If your card has only one stream, the architecture provides a PCM software mixing interface where multiple streams are mixed down to just one stream that's directed to your driver.

- Will you support your card on both little and big endian machines?

If you intend to support your chip on platforms with both types of endianness, be advised that this will impact CPU performance (on the nonnative machine) because data has to be converted whenever you read or write a register.

- Which PCM data formats will you card accept?
- What PCM rates will the card support?

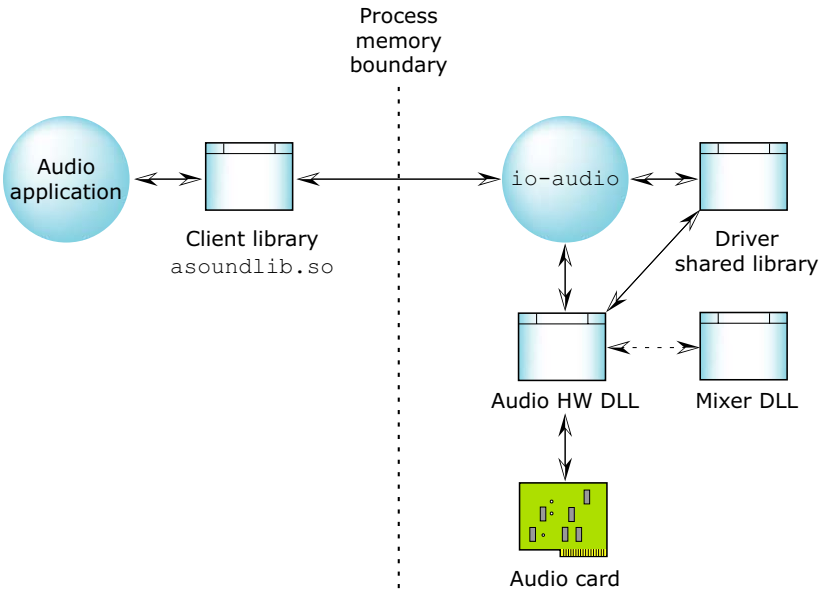
Chapter 2

Organization of a Driver

In this chapter. . .

The QNX audio system and driver

Before discussing the organization of the driver you're about to write, it's useful to first discuss the organization of the QNX audio system. Let's consider an audio application sending data to an audio card. The QNX architecture for this is:



How the driver fits into the QNX audio system.

Everything to the right of the Process memory boundary is the audio driver. To complete the audio driver for your audio card, you have to build only the Audio HW DLL.

Audio application

Produces PCM data and then makes a series of QNX Audio API calls to send this data to the Audio card. For more information, see the QNX *Audio Developer's Guide*.

Client library (**asoundlib.so**)

Translates the API calls into messages and sends them to the driver process across the fully memory-protected boundary.

In addition to this, the client library has a series of plugins that the application can use to convert its data to and from various formats. This means the driver has to support only native audio formats. The client library plugins can be used to make any necessary conversions. The mechanism by which this is done is somewhat complicated and outside the scope of this document, but it does allow for a simplified driver interface.

io-audio The main controller for all audio drivers. It's designed to support multiple audio cards simultaneously. Its primary functions are mounting and unmounting audio cards, and directing inbound messages to the correct card. For information about starting **io-audio**, see the QNX *Utilities Reference*.

Driver shared library

Provides a series of subroutines that the **io-audio** and the Audio HW DLL modules need. Due to the nature of our dynamically linked library system, the driver shared library doesn't need to be a separate file. Instead it's actually linked into the **io-audio** module, but it's still useful to think of it as a separate piece that any module can reach.

Audio hardware DLL

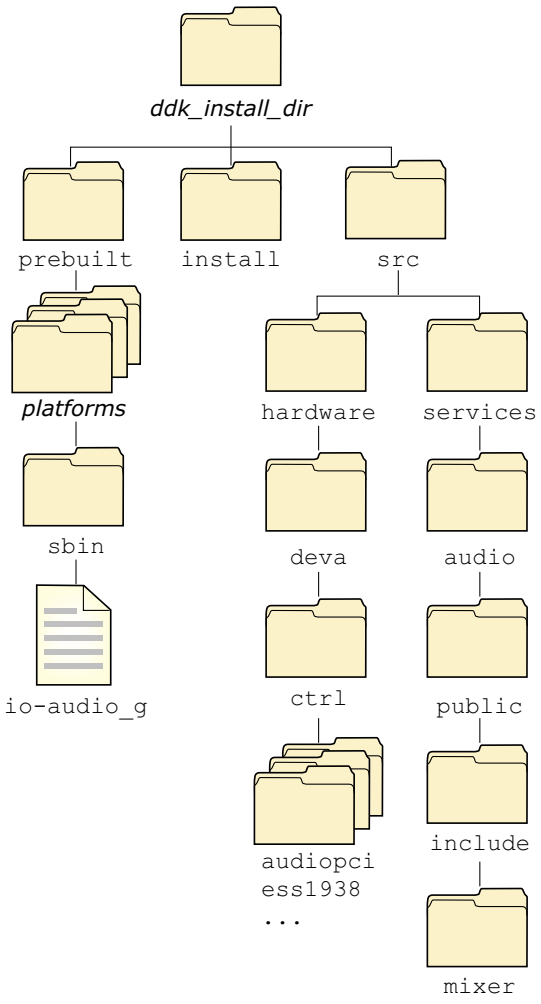
This is the piece you're developing. It's the bridge between the hardware and the rest of the audio system. Thanks to the architecture, the amount of code inside this module is quite small. The audio HW DLL module is produced as a DLL so that **io-audio**

can load and unload it at runtime so as to reduce its memory footprint.

Mixer DLL — If the card has a standard codec, you can use a mixer DLL to further reduce the work required in writing the audio HW DLL. For more information, see the Supported Codecs appendix.

DDK source code

When you install the DDK package, the source is put into a directory under the *ddk_install_dir*`/ddk-audio` directory. Currently, the directory structure for the Audio DDK looks like this:



Directory structure for the Audio DDK.

For example, the *ddk_working_dir* for the previous example would be **`~/my_DDKs/audio/`**.

Writing an Audio HW DLL

This section describes some of the basics of writing an Audio HW DLL, including:

- Opaque data types
- Custom data types
- *ctrl_init()*
- *ctrl_destroy()*
- Debugging an audio driver

Opaque data types

The API for the Audio DDK involves some structures that aren't defined in the scope of this DDK; their contents are only known to the **io-audio** layer. You typically just need to save pointers to them and pass the pointers to the functions that need them.

Here's a list of the opaque data types:

- **ado_card_t**
- **ado_dswitch_t**
- **ado_mixer_delement**
- **ado_mixer_dgroup_t**
- **ado_mixer_t**
- **ado_pcm_subchn_mixer_t**
- **ado_pcm_subchn_t**
- **ado_pcm_t**

The **ado** prefix to these names stands for **audio**.

Custom data types

Your Audio HW DLL might need to keep some internal data for its own use. The Audio DDK lets you define context-sensitive data for your hardware as well as the mixers.

To make the API more flexible (and readable), the Audio DDK uses these types that you can define as you wish:

`HW_CONTEXT_T`

Data you want to associate with the hardware.

`MIXER_CONTEXT_T`

Data you want to associate with a mixer.

By default, these types are empty structures. Use a **`#define`** directive to set these types as appropriate *before* including any of the Audio DDK header files. For example:

```
#define HW_CONTEXT_T    my_hw_context_t
#define MIXER_CONTEXT_T my_mixer_context_t
```

If you wish, you can even define **`HW_CONTEXT_T`** and **`MIXER_CONTEXT_T`** to be the same type.

The **`MIXER_CONTEXT_T`** is stored as part of the **`ado_mixer_t`** structure. If you need to access the mixer context, you need to call *ado_mixer_get_context()* because **`ado_mixer_t`** is an opaque data type.

ctrl_init()

Your Audio HW DLL must provide an entry point called *ctrl_init()*, of type **`ado_ctrl_dll_init_t`**. When **`io-audio`** loads your Audio HW DLL, it calls this function. The prototype is:

```
int32_t ctrl_init( HW_CONTEXT_T **hw_context,
                   ado_card_t *card,
                   char *args )
```

The arguments are:

hw_context	A location where your *ctrl_init()* function can store a pointer to a context-specific state structure for the card. You can define **`HW_CONTEXT_T`** to be whatever structure you want; by default, it's defined to be **`struct hw_context`**.
card	A pointer to an internal card structure, **`ado_card_t`**. This structure isn't defined in the scope of this DDK; its contents are only known to the **`io-audio`** layer. You'll need to save this pointer to pass to some functions, such as *ado_mixer_create()*, that your Audio HW DLL might call.
args	Any command-line arguments that were part of the **`io-audio`** or **`mount`** command (for more information, see the QNX *Utilities Reference*).

The first job of this initialization code is to allocate its context-specific state structure. You can allocate the hardware context by calling *ado_calloc()* or *ado_malloc()*.

Next, you need to verify that the hardware is present in the system. How you do this depends on your driver; in some cases, it isn't possible:

- If your card is a PCI device, the easiest way to attach to the card is by calling *ado_pci_device()*. This function even parses the driver arguments to find the right card to attach to.

  The *ado_pci_device()* function returns a pointer to a **`ado_pci`** structure that describes a selected PCI card. Keep a copy of this pointer in your hardware context.

- If your hardware is on another bus, you have to check the arguments, using *getsubopt()* (see the QNX *Library Reference*) to determine what hardware your driver should try to use.

You should set the short and long names that audio applications will use to identify your card's type and the specific instance of the hardware; call *ado_card_set_shortname()* and *ado_card_set_longname()*.

In addition to the above, the initialization depends on what features your Audio HW DLL supports:

- If you're writing your own audio mixer, you need to create it and its elements. See "Writing an analog mixer" in the Handling Analog Audio Data chapter.
- If you're using a standard audio mixer, you need to open its DLL. See "Using a standard mixer DLL" in the Handling Analog Audio Data chapter.
- If you want to support digital audio, you need to create a Pulse Code Modulation (PCM) device. See the Handling PCM Audio Data chapter.

If the initialization is successful, *ctrl_init()* should return 0. If an error occurs, *ctrl_init()* should return -1; in this case, **`io-audio`** unmounts the card.

ctrl_destroy()

Your Audio HW DLL must provide an entry point called *ctrl_destroy()*, of type **`ado_ctrl_dll_destroy_t`**. The **`io-audio`** manager calls *ctrl_destroy()* whenever the card is unmounted. The prototype is:

```
int32_t ctrl_destroy( HW_CONTEXT_T **hw_context );
```

This function undoes whatever you did in your *ctrl_init()* function. Typically, your *ctrl_destroy()* function:

- disconnects from the PCI device by calling *ado_pci_release()*
- frees the memory allocated for the hardware context
- does any other cleanup required.

If the cleanup is successful, *ctrl_destroy()* should return 0. If an error occurs, *ctrl_destroy()* should return -1.

Debugging an audio driver

The Audio DDK uses several constants to turn debugging messages on and off. The main one is ADO_DEBUG.

The standard Audio DDK makefiles define ADO_DEBUG if you've defined **DEBUG** in your environment. If you compile your driver with debugging options, ADO_DEBUG is defined as well to help you debug your logic.

If ADO_DEBUG is defined, several things happen:

- Error messages and warnings generated by your calls to *ado_error()* and *ado_debug()* are printed on standard output as well as being sent to the system logger, **`slogger`**. In nondebug mode, they're sent only to **`slogger`**. For more information about the system logger, see the QNX *Utilities Reference*.
- Full memory accounting is provided for all the *ado_** memory functions:
  - *ado_calloc()*,
  - *ado_free()*,
  - *ado_malloc()*,
  - *ado_realloc()*,
  - *ado_strdup()*

  You can call *ado_memory_dump()* to get a full listing of active memory. In nondebug mode, these memory functions are defined to be the standard C library functions.

If you've set ADO_DEBUG, you can also define the following macros to get specialized debug output:

- ADO_MUTEX_DEBUG — used by *ado_mutex_destroy()*, *ado_mutex_init()*, *ado_mutex_lock()*, and *ado_mutex_unlock()*.

- ADO_RWLOCK_DEBUG — used by *ado_rwlock_destroy()*, *ado_rwlock_init()*, *ado_rwlock_rdlock()*, *ado_rwlock_unlock()*, and *ado_rwlock_wrlock()*.

If you define these macros, you'll see a message whenever a lock is changed. You'll probably need to define them only if you encounter a locking problem.

If you've compiled your driver with ADO_DEBUG on, the driver won't run under the shipped **`io-audio`**, because the debugging code makes **`io-audio`** bigger, which can be a problem for embedded systems. If you want to run your driver in debug mode, use the **`io_audio_g`** that's shipped with the DDK.

Chapter 3

Handling Analog Audio Data

In this chapter. . .

The two sections, “Writing an analog mixer” and “Using a standard mixer DLL,” are mutually exclusive; you need only one of them, depending on your card evaluation.

Writing an analog mixer

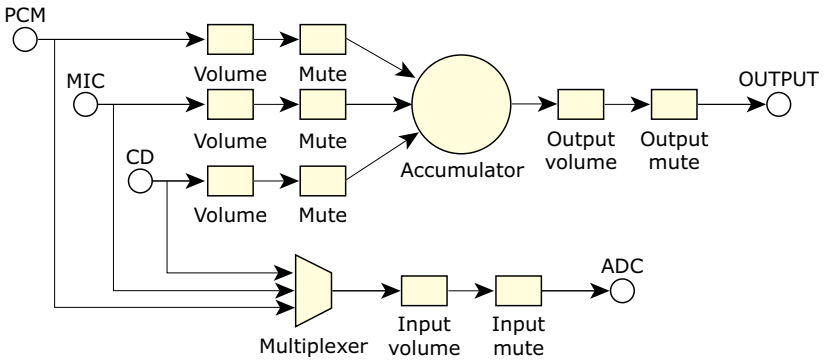
The easiest place to start in writing your Audio HW DLL is the analog mixer, because it’s the simplest part of controlling a card; it doesn’t involve any realtime constraints and small mistakes generally don’t crash the machine. When you’re using DMA, it’s possible to overwrite the kernel, so we’ll save PCM programming for later (see the Handling PCM Audio Data chapter).

In addition, if your card has an analog input (e.g. CD or Line In), it’s very easy to test the analog mixer in isolation from the rest of the sound card. In contrast, without volume controls to adjust, it’s very difficult to verify that your PCM playback (and capture) code is functioning correctly.

If your card uses one of the standard codecs (listed in the Supported Codecs appendix), see “Using a standard mixer DLL,” later in this chapter.

Mixer elements and routes

If you have a nonstandard or unsupported codec, you must define a set of mixer elements for it. A simplified codec has the following structure:



A simplified codec for an analog mixer.

In our terminology, all of the shapes are called *mixer elements*, and the lines are *mixer routes*. Some mixer elements are informational only.

The OUTPUT element is an I/O type element and holds only information such as the number of channels it contains. Other elements provide control by means of callback functions; for example, the volume elements have a callback that's used to read and set their associated gain level.

One common variation on this design is where all or some of the inputs can be mixed together into the ADC (Analog Digital Converter). This is usually done using a series of switches instead of the multiplexer.

The last important idea is that every element is routed to and from at least one other element. Only I/O elements break this rule.

The hardware design of the chip you're supporting dictates the elements and routes for the mixer. In fact, the diagram of your mixer might be similar to the example above, but is probably more complicated.

As an example, the standard AC97 diagram, has approximately 13 I/O elements and approximately 43 elements in total.

To translate the diagram to mixer software, you need to create a mixer element for every symbol on the diagram, and then create a route for every line.

Supported element types

At this point it's useful to discuss all the supported elements types, their respective attributes, any associated controls, and the function you can call to create one:

3 Dimensional Effect type 1 (3d_effect1)

Not currently in use in any driver.

Accumulator type 1 (accu1)

This element sums or adds together its input signal to produce an output signal. The number of output channels equals the input channels. For example, for stereo, all left inputs are summed to the left output and all right inputs are summed to the right output. These elements may also introduce a fixed amount of attenuation to the signal, and thus have an attenuation attribute.

Creation function: *ado_mixer_element_accu1()*

Accumulator type 2 (accu2)

This element is similar to type 1, except all input signals are summed together to a mono output. It also has an attenuation attribute.

Creation function: *ado_mixer_element_accu2()*

Accumulator type 3 (accu3)

This element is similar to type 1, except the attenuation is variable. As a result, it has a control function and an attribute of how many channels to control.

Creation function: *ado_mixer_element_accu3()*

Converter (converter)

This element converts a PCM stream from one frequency to another. It has an attribute of the bit resolution.

Input Output (io)

This element is a place holder for where a signal enters (input) or exits (output) the mixer. Typically, this a mechanical connector in the real world. This element has attributes for the number of channels and the channels that it contains. In the simple case of stereo, there are two channels: front left and front right.

Creation function: *ado_mixer_element_io()*

Multiplexer type 1 (mux1)

This element selects one of its inputs for connection to its output. In the case of multiple channels, each channel input is individually controlled. For example, in the diagram above, the multiplexer could select the left channel from the CD, and the right channel from the MIC. This element's attributes include the number of voices it controls, and a control function.

Creation function: *ado_mixer_element_mux1()*

Multiplexer type 2 (mux2)

This is a simplified type-1 multiplexer, in that it handles only mono channels.

Creation function: *ado_mixer_element_mux2()*

Pan Control type 1 (pan_control1)

Not currently in use in any driver.

Pulse Code Modulator type 1 (pcm1)

This element is a Digital to Analog Converter (DAC) for output, or an Analog to Digital Converter (ADC) for input. It's the bridging element between the analog mixer and the digital PCM sections of a soundcard. This element has an attribute that identifies which PCM device it is.

Creation function: *ado_mixer_element_pcm1()*

Pulse Code Modulator type 2 (pcm2)

This element is used when a pcm1 supports multiple subchannels. Each active subchannel is shown as a pcm2

element connected via a possible volume-and-mute element to a pcm1 element. Typically these elements are created and maintained through the *ado_pcm_subchn_mixer_create()* API function call, and not used directly when building a mixer.

Creation function: *ado_mixer_element_pcm2()*

Switch type 1 (sw1)

This element is array of simple on-or-off switches, one for every channel that the switch controls. It has a control function for setting the state of the switches. Typically, these switches are used as mute controls for streams containing more than one channel.

Creation function: *ado_mixer_element_sw1()*

Switch type 2 (sw2)

This element is a simple on-or-off switch. It has a control function to set its state. Typically, these are used as mute switches on mono channels.

Creation function: *ado_mixer_element_sw2()*

Switch type 3 (sw3)

This element is a matrix switch that controls routing of the signals it controls. To conceptualize this switch, think of a matrix with all inputs along the left side, and all outputs along the bottom. The total number of switches is thus input $\times$ outputs. This element has an attribute of the number of inputs and outputs, as well as a control function. These elements are sometimes used where a multiplexer would normally be used on the input side of the mixer to allow recording from multiple sources simultaneously.

Creation function: *ado_mixer_element_sw3()*

Tone Control type 1 (tone_control1)

Not currently in use in any driver.

Volume type 1 (volume1)

This element controls the amplitude, or gain, of analog signals that pass through it. It has attributes of the number of channels it controls, the range of gains it can control, and a control function.

Creation function: *ado_mixer_element_volume1()*

You can associate instance data with the more complex elements. If you need to access this instance data later, you have to call *ado_mixer_get_element_instance_data()* because **ado_mixer_delement_t** is an opaque data type.

Mixer groups

In the simplest terms, a *mixer group* is a collection or group of elements and associated control capabilities. For the purpose of simplifying driver coding, we further define groups as relating to either playback or capture functionality:

Playback group	Can contain up to one volume element and one mute element. Creation function: *ado_mixer_playback_group_create()*
Capture group	Can contain up to one volume, one mute, and one input selection element. Creation function: *ado_mixer_capture_group_create()*

The *input selection* element is either a multiplexer or an input switch. With these restrictions, the group control logic can be contained entirely within the **io-audio** module. To create a group, you can simply specify the group name, type, and its component elements.

Designing mixer groups

Unlike elements and routes, mixer groups aren't strictly dictated by the hardware. You, as the driver writer, can decide on the number and contents of mixer groups. In order to build a useful driver, you need to create mixer groups with a logical design that attempts to satisfy the following conditions:

- Most audio applications deal only with mixer groups, since controlling elements directly becomes quite complicated. Therefore all major operations in your mixer should be controllable through mixer groups.

  For example, the standard Photon mixer application displays and manipulates only mixer groups.

- The elements of a mixer group should control the same stream. Building a mixer group with the PCM mute and the CD volume is possible, but not really logical or useful.

- A mixer group should be associated with a PCM channel such that an audio application can control the volume with respect to the PCM channel it has open. Your Audio HW DLL needs to provide a *snd_pcm_plugin_setup()* function that returns the associated mixer group. For more information about this function, see the QNX *Audio Developer's Guide*.

- Capture mixer groups don't need to contain volume or mute controls if control of the input selection is required. In the above diagram, the PCM, MIC, and CD capture groups would contain only the multiplexer element. Another capture group would contain the input volume and input mute elements.

  It's possible to make the PCM, MIC, and CD capture groups contain the input volume and input mute elements, but this would lead application developers to believe there are independent volume and mute controls on the these inputs, when clearly they're shared.

Coding the mixer

For the purposes of demonstration, we assume that the simplified codec shown in the previous figure represents the mixer that you plan to support. The rest of this chapter demonstrates how to translate this relatively standard diagram into code.

The complete code for the sample mixer in this chapter is available in the Sample Mixer Source appendix.

Before we can write any of the mixer code, we need to get some basic requirements of the driver out of the way. We need a build environment to build this code as a DLL, and we need to provide a standard entry point for **`io-audio`** to call to initialize the chip. The easiest way to do this is to copy the Sound Blaster driver directory (**`sb`**) to a directory named for your card.

After copying the directory, you should rename the C, header, and usage-message files to something more descriptive of your chip. After doing this, make sure the code still compiles before proceeding.

Initialization

As described earlier, your Audio HW DLL must provide an entry point called *ctrl_init()*. The Organization of a Driver chapter describes the initialization that this function must do no matter what features your DLL supports.

If you're writing a custom audio mixer, the next task to perform (after *ctrl_init()* function has done the common part of the initialization) is to allocate and initialize a new **`ado_mixer_t`** structure.

Do this by calling *ado_mixer_create()*. All the information pertaining to this mixer is attached to this structure, so you need to store a copy of the returned pointer somewhere (usually in your context structure), so that you can access it later. However, **`ado_mixer_t`** is an opaque data type; your Audio HW DLL doesn't need to know what's in it.

Here's an example of initializing your Audio HW DLL if you're writing your own audio mixer:

```
int
example_mixer (ado_card_t * card, HW_CONTEXT_T * example)
```

```
{
  int32_t status;

  if ( (status = ado_mixer_create
       (card, "Example", &example->mixer, example)) != EOK )
    return (status);

  return (0);
}

ado_ctrl_dll_init_t ctrl_init;

int
ctrl_init( HW_CONTEXT_T ** hw_context, ado_card_t * card,
           char *args )
{
  example_t *example;
  if ((example = (example_t *) ado_calloc (1,
                                  sizeof (example_t))) == NULL)
  {
    ado_error ("Unable to allocate memory (%s)\n",
      strerror (errno));
    return -1;
  }

  *hw_context = example;

  /* Verify that the hardware is available here. */
  if (example_mixer(card, *hw_context) != 0)
    return -1;
  else
    return 0;
}
```

If you need to allocate memory for your mixer, you should create a cleanup function for **io-audio** to call when your mixer is destroyed. For more information, see *ado_mixer_set_destroy_func()*.

You can also create a function to be called when the mixer's hardware is reset, but this usually isn't necessary. For more information, see *ado_mixer_set_reset_func()*.

Building the mixer

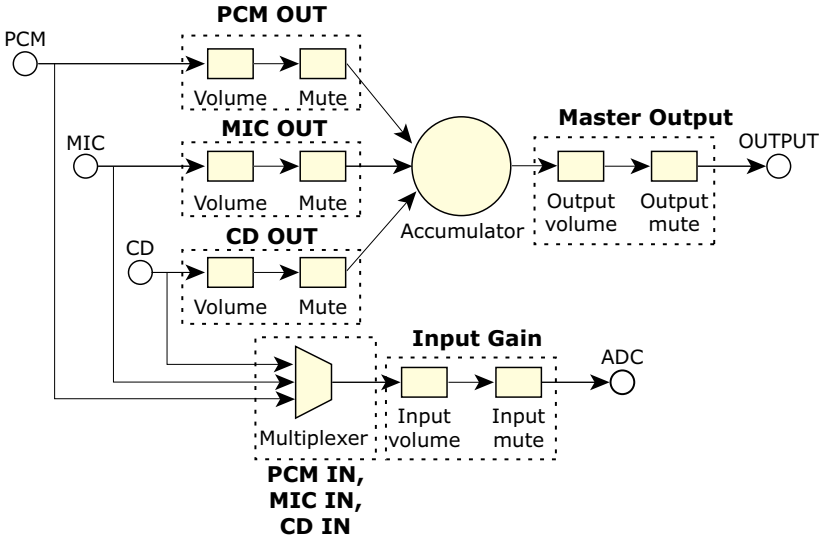
You must next construct a description of the mixer from its component parts. As mentioned earlier, a mixer consists of mixer elements, routes, and groups. In this example, there are 17 mixer

elements, 18 routes, and 8 groups. The elements and routes are relatively straightforward to identify.

Elements are any of the symbols, and routes are the paths that data can travel between them. Use the functions listed above to create the elements; use *ado_mixer_element_route_add()* to create the routes.

Don't forget to count the *point sources* and *point sinks* as elements. Though they may not be drawn as solid symbols, they are important parts of the audio architecture.

Identifying the groups is a little more troublesome. That's the reason why we enforce the rules on what can be in a group. It simplifies choosing how to divide the elements up into groups, and makes the drivers more consistent in form and behaviour. The eight groups are Master Output, Input Gain, PCM OUT, MIC OUT, CD OUT, PCM IN, MIC IN, and CD IN.



The groups in the sample analog mixer.

The PCM IN, MIC IN, and CD IN groups include the multiplexer, but specify a different input to it.

To build the mixer, first create the elements and routes, then pass pointers to the required elements to the functions that create the mixer group.

Here's the section of code that creates the master group, including all elements and routes:

```
int
build_example_mixer (MIXER_CONTEXT_T * example,
                     ado_mixer_t * mixer)
{
  int     error = 0;
  ado_mixer_delement_t *pre_elem, *elem = NULL;

  /* ################ */
  /* the OUTPUT GROUP */
  /* ################ */
  if ( (example->output_accu = ado_mixer_element_accu1
       (mixer, SND_MIXER_ELEMENT_OUTPUT_ACCU, 0)) == NULL )
    error++;

  pre_elem = example->output_accu;

  if ( !error && (elem = ado_mixer_element_volume1
       (mixer, "Output Volume", 2, output_range,
        example_master_vol_control,
        (void *) EXAMPLE_MASTER_LEFT, NULL)) == NULL)
    error++;

  if ( !error && ado_mixer_element_route_add
       (mixer, pre_elem, elem) != 0 )
    error++;

  example->master_vol = elem;
  pre_elem = elem;

  if ( !error && (elem = ado_mixer_element_sw2
       (mixer, "Output Mute", example_master_mute_control,
       (void *) EXAMPLE_MASTER_LEFT, NULL)) == NULL )
    error++;

  if ( !error && ado_mixer_element_route_add
       (mixer, pre_elem, elem) != 0 )
     error++;

  example->master_mute = elem;
```

```
  pre_elem = elem;

  if ( !error && (elem = ado_mixer_element_io
       (mixer, "Output", SND_MIXER_ETYPE_OUTPUT, 0, 2,
        stereo_voices)) == NULL )
    error++;

  if ( !error && ado_mixer_element_route_add
       (mixer, pre_elem, elem) != 0 )
    error++;

  if ( !error &&
       (example->master_grp = ado_mixer_playback_group_create
       (mixer, SND_MIXER_MASTER_OUT, SND_MIXER_CHN_MASK_STEREO,
       example->master_vol, example->master_mute)) == NULL )
    error++;

  return (0);
}
```

Don't feel that you must have all the mixer elements represented in the mixer groups. This isn't the point. The mixer elements and mixer groups are meant to be complementary. Nonstandard, complex, or just plain weird controls may not be needed at the mixer group level. They may be better as a simple mixer element or mixer switch. The mixer groups are intended to help the developer of audio applications figure out which mixer elements are related to each other and to a particular connection (e.g. PCM OUT).

In this sample mixer, none of the individual input groups (PCM IN, MIC IN, CD IN) has volume or mute controls. They're still required because they contain the capture selection switch, but the only volume and mute controls on the input side are in the Input Gain group. This is important to note because it points out that you don't need to completely fill the requirements to specify a group. If you're missing a mixer element in your hardware, you can specify NULL for the missing element, if it makes sense to group them that way.

Using a standard mixer DLL

If your card uses one of the standard codecs (listed in the Supported Codecs appendix), the amount of work you have to do is reduced.

The benefit of using standardized codecs is that you just have to write a few access functions, typically the ones that read and write the codec registers.

Before we can write these functions, we need to get some basic requirements of the driver out of the way. We need a build environment to build this code as a DLL and we need to provide a standard entry point for **`io-audio`** to call to initialize the chip.

The easiest way to do this is to copy one of the existing driver directories (**`/audio/src/hardware/deva/*`**) in the DDK to a directory named for your card or chip type. The best code to copy is either the **`template`** driver or the Sound Blaster (**`sb`**), depending on your answers to the questions in the Evaluating Your Card chapter. After copying the directory, you should rename the C, header, and use files to something more descriptive of your chip. After doing this, make sure the code still compiles before proceeding.

Initialization

As described earlier, your Audio HW DLL must provide an entry point called *ctrl_init()*. The Organization of a Driver chapter describes the initialization that this function must do no matter what features your DLL supports.

After you've verified that the hardware exists, you need to map in the card memory if it's memory-mapped and initialize a mutex in the context structure. The mutex is used to make sure only one thread is accessing the hardware registers at a given point in time. Generally you lock the mutex around any routines that access card registers.

Keep the mutex locked for as little time as possible.

Now that we have access to the hardware, the next step is to inform the upper layers of the driver of the capabilities of this hardware. We

do this by creating devices: mixers and PCM channels. We'll look at creating the PCM device in the next chapter.

Since we have a standard codec, we use the *ado_mixer_dll()* function to create the mixer structure and load the appropriate mixer DLL. The prototype is:

```
int32_t ado_mixer_dll( ado_card_t *card,
                       char *mixer_dll,
                       uint32_t version,
                       void *params,
                       void *callbacks,
                       ado_mixer_t **rmixer );
```

The arguments to *ado_mixer_dll()* include:

- the card pointer, so that the function knows what card to attach the new mixer to
- the name and version of the DLL to load
- structures for defining necessary callback functions
- a location where the function can store a pointer to the internal mixer structure.

The data types and contents of the *params* and *callbacks* structures depend on the mixer DLL that you're loading; see the Supported Codecs appendix for details.

The *params* structure is the key to making the mixer work correctly. It tells the mixer DLL about functions that you've written in your Audio HW DLL, typically to read and write the codec registers. This structure contains pointers to a **`hw_context`** structure and (typically) functions that read and write the codec registers. The **`hw_context`** is generally, but it doesn't need to be, the same context that you allocated at the beginning of the *ctrl_init()* function. The **`hw_context`** is passed back to you as a parameter when the mixer DLL calls the read or write routines.

Be sure to thoroughly test the callbacks that read and write the codec registers. If they don't work correctly, the mixer DLL might misbehave or fail.

The *callbacks* structure tells you about functions that are defined in the mixer DLL that your Audio HW DLL needs to call in order to control the device. The *ado_mixer_dll()* function fills in this structure, based on the mixer DLL that you're opening.

Testing the code

To test this code, start up the driver and input an analog signal to one of the codec inputs (line, CD, etc.). Then, using the GUI mixer, try to control the volume of that signal at the speakers. Once this works reliably, you can move onto the next chapter.

Chapter 4

Handling PCM Audio Data

In this chapter. . .

What's a PCM device?

In this architecture, a PCM device is a device capable of supporting either a PCM capture channel, or a PCM playback channel, or both. A PCM capture channel converts an analog signal to a digital PCM stream, whereas a PCM playback channel takes a digital PCM stream and converts it to analog. A PCM device may also support converting multiple PCM streams simultaneously; each of these streams is called a PCM *subchannel*.

Creating a PCM device

As described earlier, your Audio HW DLL must provide an entry point called *ctrl_init()*. The Organization of a Driver chapter describes the initialization that this function must do no matter what features your DLL supports.

In much the same way that we created a mixer device in the previous chapter, we now create a PCM device using the *ado_pcm_create()* function. This informs the upper levels of software that this card now supports a PCM device. If you call this function again, it creates additional devices.

ado_pcm_create()

The prototype of the *ado_pcm_create()* function is:

```
int32_t ado_pcm_create( ado_card_t *card,
                        char *name,
                        uint32_t flags,
                        char *id,
                        uint32_t play_subchns,
                        ado_pcm_cap_t *play_cap,
                        ado_pcm_hw_t *play_hw,
                        uint32_t cap_subchns,
                        ado_pcm_cap_t *cap_cap,
                        ado_pcm_hw_t *cap_hw,
                        ado_pcm_t **rpcm );
```

The arguments are:

card The *card* argument that **io-audio** passed to your Audio HW DLL's *ctrl_init()* function. The library uses this argument as the card to link the new device onto.

name, id Text names for the PCM device, usually a variation of the card name. They're used only for information display by client applications.

flags Information about the device for the use of client applications, for example, indicating whether or not both the playback and capture can be used at the same time. These flags are informational only; enforcing any of these conditions is done by code and isn't based on these flags.

play_subchns, *cap_subchns*

The number of playback and capture subchannels supported. If zero, the channel isn't supported.

play_cap, cap_cap

Structures listing the full capabilities of the device's playback and capture channels. These include *format*, *rate*, and *voices* supported. The upper driver layers use this information to verify a client request before allowing the request to pass to the hardware. This information is also passed back to the client as the static capabilities of the device.

play_hw, *cap_hw*

Structures containing the playback and capture callback functions to be called by the upper layers of the driver. It's in these functions that you actually program the hardware. For more information, see "How does the PCM stream operate?" later in this chapter.

rpcm The location in which to store a pointer to the internal PCM device structure. You'll need this pointer for additional PCM function calls.

How does the PCM stream operate?

In order to make the PCM device work, you need to define the callback functions in the **`ado_pcm_hw_t`** structures for the capture and playback portions of the PCM device.

Before we look at them in detail, let's first review how a PCM stream operates in hardware. The model used in this architecture is a DMA buffer in memory that's divided into two or more buffer fragments. When instructed to do so, the hardware acts on a fragment using DMA, and then generates an interrupt on completing the fragment.

So, if we consider the simplified case of playback with a 50K buffer, composed of two fragments, here's what happens when the client application sends data:

- First the data is written into the DMA buffer until both fragments are filled, then the hardware is told to start playing the first fragment, and to continue playing the fragments in order until told to stop.
- When the hardware signals an interrupt, meaning the first fragment has been completed, **`io-audio`** puts more client data into the first fragment, and the cycle continues.
- The driver stops the playback when the next fragment to be played isn't filled with new data. This is called an *underrun* condition.

From a programming perspective, if the hardware can be set up to do a looping DMA buffer playback with an interrupt every *x* bytes, implementing this model is very straightforward. A variation on this theme is to reprogram the DMA engine after every fragment in the interrupt routine. In the general case, the client suggests the fragment size and number of fragments, but the driver has the ultimate authority on these parameters.

Chapter 5

API Reference

This chapter describes the API data types and functions in alphabetical order. (The **ado** prefix to these names stands for **audio**.)

ado_attach_interrupt()
: Attach a handler function to an interrupt

ado_calloc()
: Allocate space for an array

ado_card_set_longname()
: Create a name that identifies an instance of the audio hardware

ado_card_set_shortname()
: Create a name that identifies the type of the audio hardware

ado_debug()
: Send a debugging string to **slogger** for logging

ado_device_mmap()
: Map a region of physical memory

ado_device_munmap()
: Release a virtual memory region

ado_error()
: Send an error string to **slogger** for logging

ado_free()
: Deallocate a block of memory

ado_malloc()
: Allocate memory

ado_memory_dump()
: Show all memory currently in use by the *ado_** family of allocation functions

ado_mixer_capture_group_create()
: Create a capture group

ado_mixer_create()
: Allocate a new mixer structure, then attach it

ado_mixer_dll() Load the specified standard mixer DLL

ado_mixer_element_accu1()

Create an accumulator (type 1) element

ado_mixer_element_accu2()

Create an accumulator (type 2) element

ado_mixer_element_accu3()

Create an accumulator (type 3) element

ado_mixer_element_io()

Create an input/output element

ado_mixer_element_mux1()

Create a multiplexer (type 1) element

ado_mixer_element_mux2()

Create a multiplexer (type 2) element

ado_mixer_element_notify()

Notify the upper driver levels that there's been a change in an element

ado_mixer_element_pcm1()

Create a PCM (type 1) element

ado_mixer_element_pcm2()

Create a PCM (type 2) element

ado_mixer_element_route_add()

Indicate an ordered relationship between two mixer elements

ado_mixer_element_sw1()

Create a switch (type 1) element

ado_mixer_element_sw2()

Create a switch (type 2) element

ado_mixer_element_sw3()

Create a switch (type 3) element

ado_mixer_element_vol_range_max()

Read the maximum volume setting of the mixer volume element

ado_mixer_element_vol_range_min()

Read the minimum volume setting of the mixer volume element

ado_mixer_element_volume1()

Create a volume (type 1) element

ado_mixer_find_element()

Search a mixer for an element

ado_mixer_find_group()

Search a mixer for a group

ado_mixer_get_context()

Get a pointer to a mixer's context structure

ado_mixer_get_element_instance_data()

Access a mixer element's instance data

ado_mixer_lock()

Limit access to the mixer

ado_mixer_playback_group_create()

Create a playback group

ado_mixer_set_destroy_func()

Attach a function to the mixer to be called when the mixer is destroyed

ado_mixer_set_name()

Attach the character string name to the mixer

ado_mixer_set_reset_func()

Attach a function to the mixer to be called when a hardware reset of the mixer occurs

ado_mixer_switch_new()

Create a new mixer switch

ado_mixer_unlock()

Unlock the attribute structure

ado_mutex_destroy()

Destroy a mutex

ado_mutex_init()

Initialize a mutex

ado_mutex_lock()

Lock a mutex

ado_mutex_unlock()

Unlock a mutex

`ado_pci` Data structure that describes a selected PCI card

ado_pci_device()

Try to connect to a specified PCI card

ado_pci_release()

Detach from a given PCI card

`ado_pcm_cap_t`

Data structure of capabilities of a PCM device

`ado_pcm_config_t`

Data structure that describes the configuration of a PCM subchannel

ado_pcm_chn_mixer()

Logically associate a mixer element and group with a PCM device

ado_pcm_create()

Create a PCM audio device

ado_pcm_dma_int_size()

Obtain the fragment size of a PCM channel

ado_pcm_format_bit_width()

Obtain the sample width, in bits, for a given format

ado_pcm_hw_t

Data structure of callbacks for PCM devices

ado_pcm_subchn_caps()

Get a pointer to the capabilities structure for a subchannel

ado_pcm_subchn_is_channel()

Check if a channel is a subchannel of a PCM device

ado_pcm_subchn_mixer()

Logically associate a mixer element and group with a PCM *subchannel* device

ado_pcm_subchn_mixer_create()

Create a PCM subchannel mixer

ado_pcm_subchn_mixer_destroy()

Destroy a PCM subchannel mixer

ado_pcm_sw_mix()

Provide a mechanism for an audio chip to support multiple simultaneous streams

ado_realloc() Allocate, reallocate, or free a block of memory

ado_rwlock_destroy()

Destroy a read-write lock

ado_rwlock_init()

Initialize a read-write lock

ado_rwlock_rdlock()

Acquire a shared read lock on a read-write lock

ado_rwlock_unlock()

Unlock a read-write lock

ado_rwlock_wrlock()

Acquire an exclusive write lock on a read-write lock

ado_shm_alloc()

Allocate shared memory

ado_shm_free() Release memory and unlink shared memory

ado_shm_mmap()

Map a shared memory region into the Audio HW DLL's address space

ado_strdup() Create a duplicate of a string

dma_interrupt() Signal the upper layers of a driver that the current fragment of a subchannel has been completed by the DMA engine

ado_attach_interrupt()

Attach a handler function to an interrupt

Synopsis:

```
#include <audio_driver.h>

int32_t ado_attach_interrupt (
    ado_card_t *card,
    int32_t irq,
    void (*handler) (
              HW_CONTEXT_T *hw_context,
              int32_t irqnum ),
    HW_CONTEXT_T *hw_context );
```

Arguments:

card	The *card* argument that **io-audio** passed to your Audio HW DLL's *ctrl_init()* function (see the Organization of a Driver chapter).
irq	The interrupt request number.
handler	The handler function. When the *handler* is invoked, the *hw_context* and the IRQ number are passed in as arguments.
hw_context	A pointer to your Audio HW DLL's context-specific data, which you created in the *ctrl_init()* function.

Description:

The *ado_attach_interrupt()* function attaches a handler function to an interrupt.

The **io-audio** manager automatically detaches any interrupt handlers when your Audio HW DLL is unmounted.

Returns:

0 on success, or a negative number if an error occurred (*errno* is set).

Errors:

EAGAIN Insufficient system resources to create a thread.

Classification:

QNX Neutrino

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

ado_calloc()

Allocate space for an array

Synopsis:

```
#include <audio_driver.h>

void *ado_calloc( size_t n,
                  size_t size );
```

Arguments:

n The number of elements to allocate in the array.

size The size of each element.

Description:

The *ado_calloc()* macro allocates space from the heap for an array.

This macro is defined as *ado_calloc_debug()*, or *calloc()*, if ADO_DEBUG is defined; see "Debugging an audio driver" in the Organization of a Driver chapter.

The advantage of using the debug variant is that it tracks the memory allocated until it's freed; see *ado_memory_dump()*.

Returns:

The same as *calloc()*: a pointer to the start of the allocated memory, or NULL if there's insufficient memory available or if *size* is zero.

Classification:

QNX Neutrino

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

See also:

ado_free(), *ado_malloc()*, *ado_memory_dump()*, *ado_realloc()*

calloc() in the QNX *Library Reference*

ado_card_set_longname()

Create a name that identifies an instance of the audio hardware

Synopsis:

```
#include <audio_driver.h>

void ado_card_set_longname( ado_card_t *card,
                            char *name,
                            uint32_t addr );
```

Arguments:

card The *card* argument that **io-audio** passed to your Audio HW DLL's *ctrl_init()* function (see the Organization of a Driver chapter).

name This string is generally, but doesn't have to be, the card's short name; see *ado_card_set_shortname()*.

addr A number that's unique to this instance of the DLL, such as the address of the chip or I/O port.

Description:

The *ado_card_set_longname()* convenience function builds a character string of up to 80 characters that identifies an instance of the audio hardware.

The resulting name must be unique, even across multiple invocations of the same driver. For example, **SoundBlaster16 @ 220**.

Classification:

QNX Neutrino

Safety	
Cancellation point	No
Interrupt handler	No
Signal handler	No

continued...

Safety	
Thread	No

See also:

ado_card_set_shortname()

ado_card_set_shortname()

Create a name that identifies the type of the audio hardware

Synopsis:

```
#include <audio_driver.h>

void ado_card_set_shortname( ado_card_t *card,
                             char *name );
```

Arguments:

card The *card* argument that **io-audio** passed to your Audio HW DLL's *ctrl_init()* function (see the Organization of a Driver chapter).

name The short name for the card. This name is expected to enable the user to distinguish between different makes and models of sound hardware, but isn't expected to be unique across multiple invocations of the same driver. For example, **SoundBlaster16**.

☞ If an application needs to distinguish between invocations of the same driver, it should use the card's long name; see *ado_card_set_longname()*.

Description:

The *ado_card_set_shortname()* convenience function stores a character string of up to 32 characters that identifies the type of the audio hardware.

Classification:

QNX Neutrino

Safety	
Cancellation point	No
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	No

See also:

ado_card_set_longname()

ado_debug()

Send a debugging string to **slogger** *for logging*

Synopsis:

```
#include <audio_driver.h>

void ado_debug( uint32_t lvl,
                char *format,
                ... );
```

Arguments:

lvl The debugging level. If the global variable *db_lvl* logically ANDed with *lvl* is true, *ado_debug()* sends the debugging string to the system logger.

The *db_lvl* global variable is set based on the **-v[v...]** option to **io-audio**.

format A formatting string, similar to that used by *printf()*.

... Additional arguments, as required by the *format* string.

Description:

The *ado_debug()* sends a debugging string to **slogger** for logging.

If ADO_DEBUG is defined, these messages are also sent to standard output. For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Classification:

QNX Neutrino

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

See also:

ado_error()

io-audio, **slogger** in the QNX *Utilities Reference*

printf() in the QNX *Library Reference*

ado_device_mmap()

Map a region of physical memory

Synopsis:

```
#include <audio_driver.h>

void *ado_device_mmap ( unsigned long addr,
                        unsigned long size );
```

Arguments:

addr The address of the physical memory region.

size The size of the physical memory region, in bytes.

Description:

The *ado_device_mmap()* function maps the specified region of physical memory into the driver's virtual memory.

Returns:

A pointer to the resultant virtual memory region, or MAP_FAILED if an error occurred (*errno* is set).

Errors:

ENOMEM The address range requested is outside of the allowed process address range, or there wasn't enough memory to satisfy the request.

ENXIO The address from *addr* for *size* bytes is invalid for the requested object.

Classification:

QNX Neutrino

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

See also:

ado_device_munmap()

ado_device_munmap()

Release a virtual memory region

Synopsis:

```
#include <audio_driver.h>

int ado_device_munmap( void *addr,
                       unsigned long size );
```

Arguments:

addr The address of the virtual memory region.

size The size of the virtual memory region, in bytes.

Description:

The *ado_device_munmap()* function releases any virtual memory regions that were mapped into the region, starting at *addr* that runs for *size* bytes, rounded up to the next multiple of the page size. Subsequent references to these pages cause a SIGSEGV signal to be set on the process.

If there are no mappings in the specified address range, calling *ado_device_munmap()* has no effect.

Returns:

-1 if an error occurred (*errno* is set). Any other value indicates success.

Errors:

EINVAL The addresses in the specified range are outside the range allowed for the address space of a process.

ENOSYS Memory unmapping isn't supported by this implementation.

Classification:

QNX Neutrino

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

See also:

ado_device_mmap()

ado_error()

Send an error string to **slogger** *for logging*

Synopsis:

```
#include <audio_driver.h>

void ado_error( char *format, ... );
```

Arguments:

format	A formatting string, similar to that used by *printf()*.
...	Additional arguments, as required by the *format* string.

Description:

This function sends an error string to **slogger** for logging.

If ADO_DEBUG is defined, these messages are also sent to standard output. For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Classification:

QNX Neutrino

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

See also:

ado_debug()

slogger in the QNX *Utilities Reference*

printf() in the QNX *Library Reference*

ado_free()

Deallocate a block of memory

Synopsis:

```
#include <audio_driver.h>

void ado_free( void *ptr );
```

Arguments:

ptr A pointer to the block of memory to be freed.

Description:

The *ado_free()* macro deallocates the given block of memory that you allocated by calling *ado_calloc()*, *ado_malloc()*, *ado_realloc()*, or *ado_strdup()*.

The *ado_free()* macro is defined as *ado_free_debug()*, or *free()*, if ADO_DEBUG is defined; see "Debugging an audio driver" in the Organization of a Driver chapter.

The advantage of using the debug variants of the memory functions is that they track the memory allocated; see *ado_memory_dump()*. The debug variant of *ado_free()* stops this tracking for the given block.

Classification:

QNX Neutrino

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

See also:

ado_calloc(), *ado_malloc()*, *ado_memory_dump()*, *ado_realloc()*, *ado_strdup()*

free() in the QNX *Library Reference*

ado_malloc()

Allocate memory

Synopsis:

```
#include <audio_driver.h>

void *ado_malloc( size_t size );
```

Arguments:

size The amount of memory to allocate, in bytes.

Description:

The *ado_malloc()* macro allocates a block of memory of *size* bytes.

This macro is defined as *ado_malloc_debug()*, or *malloc()*, if ADO_DEBUG is defined; see "Debugging an audio driver" in the Organization of a Driver chapter.

The advantage of using the debug variant is that it tracks the memory allocated until it's freed; see *ado_memory_dump()*.

Returns:

A pointer to the memory object that was allocated, or NULL if there wasn't enough memory available.

Classification:

QNX Neutrino

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

See also:

ado_calloc(), *ado_free()*, *ado_memory_dump()*, *ado_realloc()*

malloc() in the QNX *Library Reference*

ado_memory_dump()

Show all memory currently in use by the ado_* family of allocation functions

Synopsis:

```
#include <audio_driver.h>

void ado_memory_dump( void );
```

Description:

If ADO_DEBUG is defined, *ado_memory_dump()* prints on standard output all the memory currently in use by the *ado_*()* family of allocation functions. This is a very handy debugging tool for memory leaks.

When ADO_DEBUG isn't defined, the *ado_*()* allocation functions are simply redefined to the standard functions so as not to incur any speed penalties.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Classification:

QNX Neutrino

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

See also:

ado_calloc(), *ado_free()*, *ado_malloc()*, *ado_realloc()*, *ado_strdup()*

ado_mixer_capture_group_create()

Create a capture group

Synopsis:

```
#include <audio_driver.h>

ado_mixer_dgroup_t *ado_mixer_capture_group_create
  ( ado_mixer_t *mixer,
    char *name,
    uint32_t channels,
    ado_mixer_delement_t *vol_elem,
    ado_mixer_delement_t *mute_elem,
    ado_mixer_delement_t *cap_elem,
    ado_mixer_delement_t *mux_in_elem );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the group in. This structure was created by *ado_mixer_create()*.

name — The name of the group, which can be up to 31 characters long. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

channels — A bitmap of the channels in the group; any combination of:

- SND_MIXER_CHN_MASK_MONO
- SND_MIXER_CHN_MASK_FRONT_LEFT
- SND_MIXER_CHN_MASK_FRONT_RIGHT
- SND_MIXER_CHN_MASK_FRONT_CENTER
- SND_MIXER_CHN_MASK_REAR_LEFT
- SND_MIXER_CHN_MASK_REAR_RIGHT
- SND_MIXER_CHN_MASK_WOOFER
- SND_MIXER_CHN_MASK_STEREO

vol_elem — The volume element for the group.

mute_elem	The mute element for the group.
cap_elem	The capture element for the group.
mux_in_elem	If the *cap_elem* is a multiplexer, the multiplexer takes its input from this element when capturing. This is the element that's immediately upstream from the multiplexer.

Description:

The *ado_mixer_capture_group_create()* function automates the allocation and filling of an **`ado_mixer_dgroup_t`** structure representing a channel in the capture direction.

Returns:

A pointer to the newly created capture group.

Classification:

QNX Neutrino

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

See also:

ado_mixer_playback_group_create()

ado_mixer_create()

Allocate a new mixer structure, then attach it

Synopsis:

```
#include <audio_driver.h>

int32_t ado_mixer_create
  ( ado_card_t *card,
    char *id,
    ado_mixer_t **rmixer,
    MIXER_CONTEXT_T *context );
```

Arguments:

card The *card* argument that **io-audio** passed to your Audio HW DLL's *ctrl_init()* function (see the Organization of a Driver chapter).

id The name of the mixer. This can be a maximum of 63 characters.

rmixer A location where *ado_mixer_create()* can store a pointer to the new mixer structure. The **ado_mixer_t** structure is opaque to your Audio HW DLL, but you need to pass a pointer to it to the other mixer functions. You typically save this pointer in your Audio HW DLL's hardware context structure.

context A pointer to contextual data that you want to associate with the mixer. You can define **MIXER_CONTEXT_T** to be whatever structure you want; by default, it's defined to be **struct mixer_context**. See "Custom data types" in the Organization of a Driver chapter.

Description:

The *ado_mixer_create()* function allocates a new mixer structure and attaches it to the provided *card* and *context*.

Returns:

-1 if an error occurred (*errno* is set). Any other value indicates success.

Errors:

ENOMEM Not enough free memory to create a new mixer.

Classification:

QNX Neutrino

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

See also:

ctrl_init() in the Organization of a Driver chapter, "Writing an analog mixer" in the Handling Analog Audio Data chapter

ado_mixer_dll()

Load the specified standard mixer DLL

Synopsis:

```
#include <audio_driver.h>

int32_t ado_mixer_dll( ado_card_t *card,
                       char *mixer_dll,
                       uint32_t version,
                       void *params,
                       void *callbacks,
                       ado_mixer_t **rmixer );
```

Arguments:

card — The *card* argument that **io-audio** passed to your Audio HW DLL's *ctrl_init()* function (see the Organization of a Driver chapter).

mixer_dll — The name of the standard mixer DLL.

This argument is only the unique part of the mixer DLL's file name. For example, if the mixer DLL's filename is **audio-mixer-ac97.so**, you should specify *mixer_dll* as **ac97**.

version — The version of the mixer DLL; currently not used.

params — A pointer to configuration information and control functions that the mixer DLL needs and that your Audio HW DLL must provide (for example, to read and write the codec registers). This structure is defined in the header file for the DLL e.g. **<mixer/***mixer_name***_dll.h>**

For example, if you're loading the Audio Codec 97 (AC97), set up a structure of type **ado_mixer_dll_params_ac97_t** and pass a pointer to it as the *params* argument to this function.

callbacks — A location that *ado_mixer_dll()* fills in with a list of functions, provided by the mixer DLL, that your Audio HX DLL can use to control special mixer functions,

such as sample rate conversion. Consult the documentation for the specific mixer DLL.

For example, if you're loading AC97, create a structure of type **`ado_mixer_dll_callbacks_ac97_t`** and pass a pointer to it as the *callbacks* argument to this function.

rmixer A location where *ado_mixer_dll()* can store a pointer to the new mixer structure. The **`ado_mixer_t`** structure is opaque to your Audio HW DLL, but you need to pass a pointer to it to the other mixer functions. You typically save this pointer in your Audio HW DLL's hardware context.

Description:

The *ado_mixer_dll()* function loads the specified standard *mixer_dll* and returns a pointer to a newly allocated mixer structure.

Returns:

-1 if an error occurred (*errno* is set). Any other value indicates success.

Errors:

ENOMEM Not enough free memory to create a new mixer.

ELIBACC The call to *dlopen()* failed for the mixer DLL specified.

ELIBBAD The call to *dlsym()* failed for the mixer DLL specified.

EPROGMISMATCH

The **ADO_MAJOR_VERSION** of the mixer DLL doesn't match with **`io-audio`**'s.

Classification:

QNX Neutrino

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

See also:

ctrl_init() in the Organization of a Driver chapter, "Writing an analog mixer" in the Handling Analog Audio Data chapter, Supported Codecs appendix

ado_mixer_element_accu1()

Create an accumulator (type 1) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_accu1 (
     ado_mixer_t *mixer,
     char *name,
     int32_t attenuation );
```

Arguments:

mixer	A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.
name	The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.
attenuation	The amount of attenuation the element contributes to the stream passing through it; usually 0 dB.

Description:

The *ado_mixer_element_accu1()* convenience function automates the creation of an accumulator (type 1) element. An accumulator of type 1 mixes together multiple inputs into a single output.

Returns:

A pointer to the newly allocated accumulator (type 1) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), ado_mixer_element_accu2(), ado_mixer_element_accu3()

ado_mixer_element_accu2()

Create an accumulator (type 2) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_accu2(
    ado_mixer_t *mixer,
    char *name,
    int32_t attenuation );
```

Arguments:

mixer	A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.
name	The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.
attenuation	The amount of attenuation the element contributes to the stream passing through it; usually 0 dB.

Description:

The *ado_mixer_element_accu2()* convenience function automates the creation of an accumulator (type 2) element. An accumulator of type 2 mixes together multiple inputs into a single MONO output.

Returns:

A pointer to the newly allocated accumulator (type 2) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), ado_mixer_element_accu1(), ado_mixer_element_accu3()

ado_mixer_element_accu3()

Create an accumulator (type 3) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *
   ado_mixer_element_accu3 (
      ado_mixer_t *mixer,
      char *name,
      uint32_t number_of_voices,
      struct snd_mixer_element_accu3_range *ranges,
      ado_mixer_delement_control_accu3_t *control,
      void *instance_data,
      void (*instance_free) (void *data) );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name — The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

attenuation — The amount of attenuation the element contributes to the stream passing through it; usually 0 dB.

number_of_voices — The number of voices in each channel that the element accumulates.

ranges — The range of the attenuation control for the type-3 accumulator.

control — A callback function, of type **ado_mixer_delement_control_accu3_t**, that determines the attenuation.

The prototype is:

```
int32_t control(
            MIXER_CONTEXT_T *context,
            ado_mixer_delement_t *element,
            uint8_t set,
            uint32_t *voices,
            void *instance_data );
```

instance_data A pointer to any instance data that you need to pass to the *control* callback. This can be a pointer to allocated memory, in which case you'll need to define the *instance_free* function.

If you need to access this instance data, you have to call *ado_mixer_get_element_instance_data()* because **ado_mixer_delement_t** is an opaque data type.

instance_free A function that must free any allocated instance data. It's called when the element is destroyed.

Description:

The *ado_mixer_element_accu3()* convenience function automates the creation of an accumulator (type 3) element. An accumulator of type 3 mixes together multiple inputs into a single output with programmable attenuation.

Returns:

A pointer to the newly allocated accumulator (type 3) element.

Classification:

QNX Neutrino

Safety	
Cancellation point	No

continued...

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

ado_mixer_create(), *ado_mixer_element_accu1()*, *ado_mixer_element_accu2()*, *ado_mixer_get_element_instance_data()*

ado_mixer_element_io()

Create an input/output element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_io
  ( ado_mixer_t *mixer,
    char *name,
    int32_t type,
    uint32_t attrib,
    uint32_t number_of_voices,
    snd_mixer_voice_t *voices );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

type The type of element; one of:

- SND_MIXER_ETYPE_INPUT
- SND_MIXER_ETYPE_OUTPUT

attrib Currently not used; set it to 0.

number_of_voices

The number of voices passing through the element.

voices An array of **snd_mixer_voice_t** structures (see below). Each entry describes one of the voices.

Description:

The *ado_mixer_element_io()* convenience function automates the creation of an input/output element. An input/output element is usually a physical connector on the sound card.

The **snd_mixer_voice_t** structure is defined as:

```
typedef struct
{
    uint16_t   voice:15, vindex:1;
    uint8_t    reserved[124];
} snd_mixer_voice_t;
```

The members include:

voice One of:

- SND_MIXER_VOICE_UNUSED
- SND_MIXER_VOICE_MONO
- SND_MIXER_VOICE_LEFT
- SND_MIXER_VOICE_RIGHT
- SND_MIXER_VOICE_CENTER
- SND_MIXER_VOICE_REAR_LEFT
- SND_MIXER_VOICE_REAR_RIGHT
- SND_MIXER_VOICE_WOOFER

vindex Not used; set this to 0.

Returns:

A pointer to the newly allocated input/output element.

Classification:

QNX Neutrino

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

ado_mixer_element_mux1()

Create a multiplexer (type 1) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_mux1 (
    ado_mixer_t *mixer,
    char *name,
    uint32_t attrib,
    uint32_t number_of_voices,
    ado_mixer_delement_control_mux1_t *control,
    void *instance_data,
    void (*instance_free) (void *data) );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name — The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

attrib — Not currently used; set it to 0.

number_of_voices — The number of voices passing through the element.

control — A callback function, of type **ado_mixer_delement_control_mux1_t**, that controls the channels.

The prototype is:

```
int32_t control(
            MIXER_CONTEXT_T *context,
            ado_mixer_delement_t *element,
            uint8_t set,
            ado_mixer_delement_t **inelements,
            void *instance_data );
```

instance_data — A pointer to any instance data that you need to pass to the *control* callback. This can be a pointer to allocated memory, in which case you'll need to define the *instance_free* function.

If you need to access this instance data, you have to call *ado_mixer_get_element_instance_data()* because **`ado_mixer_delement_t`** is an opaque data type.

instance_free — A function that must free any allocated instance data. It's called when the element is destroyed.

Description:

The *ado_mixer_element_mux1()* convenience function automates the creation of a multiplexer (type 1) element. A multiplexer of type 1 selects exactly one of many inputs to be routed to its output. Selecting a new input automatically deselects the previous one, so that only one input is selected.

Returns:

A pointer to the newly allocated multiplexer (type 1) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), *ado_mixer_element_mux2()*, *ado_mixer_get_element_instance_data()*

ado_mixer_element_mux2()

Create a multiplexer (type 2) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_mux2 (
    ado_mixer_t *mixer,
    char *name,
    uint32_t attrib,
    ado_mixer_delement_control_mux2_t *control,
    void *instance_data,
    void (*instance_free) (void *data) );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name — The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

attrib — Not currently used; set it to 0.

control — A callback function, of type **ado_mixer_delement_control_mux2_t**, that controls the channels.

The prototype is:

```
int32_t control(
            MIXER_CONTEXT_T *context,
            ado_mixer_delement_t *element,
            uint8_t set,
            ado_mixer_delement_t **inelements,
            void *instance_data );
```

instance_data — A pointer to any instance data that the *control* callback might need. This can be a pointer to

allocated memory, in which case you'll need to define the *instance_free* function.

If you need to access this instance data, you have to call *ado_mixer_get_element_instance_data()* because **`ado_mixer_delement_t`** is an opaque data type.

instance_free A function that must free any allocated instance data. It's called when the element is destroyed.

Description:

The *ado_mixer_element_mux2()* convenience function automates the creation of a multiplexer (type 2) element. A multiplexer of type 2 selects zero or more inputs to be routed to its output. Each input has its own selector, which is independent.

Returns:

A pointer to the newly allocated multiplexer (type 2) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), *ado_mixer_element_mux1()*, *ado_mixer_get_element_instance_data()*

ado_mixer_element_notify()

Notify the upper driver levels that there's been a change in an element

Synopsis:

```
#include <audio_driver.h>

void ado_mixer_element_notify
  ( ado_mixer_t *mixer,
    ado_mixer_delement_t *delement,
    uint32_t cmd,
    ado_ocb_t *ocb );
```

Arguments:

mixer	A pointer to the **ado_mixer_t** structure that specifies the mixer that the element belongs to. This structure was created by *ado_mixer_create()*.
delement	A pointer to the element.
cmd	The type of change. Possible changes include the addition or removal of an element, and a change in an element's values; see SND_MIXER_READ_* in **<sys/asound.h>**
ocb	The OCB of the application making the change. This application isn't notified, because it presumably is already aware of the change. If *ocb* is NULL, all applications, including the one making the change, are notified.

Description:

The *ado_mixer_element_notify()* function is used to notify the higher levels of abstraction that there's been a change in an element.

This notification is passed up the chain, through the mixer groups associated with this element, eventually reaching user applications.

Classification:

QNX Neutrino

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

ado_mixer_element_pcm1()

Create a PCM (type 1) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_pcm1
  ( ado_mixer_t *mixer,
    char *name,
    int32_t type,
    uint32_t number_of_devices,
    int32_t *devices );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

type The type of the element; one of:

- SND_MIXER_ETYPE_CAPTURE1
- SND_MIXER_ETYPE_PLAYBACK1

number_of_devices

The number of physical devices used to make this element (usually 1).

devices An array of device numbers that identify the physical devices used.

Description:

The *ado_mixer_element_pcm1()* convenience function automates the creation of a PCM (type 1) element. A PCM type 1 element specifies the start of a playback channel.

Returns:

A pointer to the newly allocated PCM (type 1) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), *ado_mixer_element_pcm2()*

ado_mixer_element_pcm2()

Create a PCM (type 2) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_pcm2
  ( ado_mixer_t *mixer,
    char *name,
    uint32_t type,
    int32_t device,
    int32_t subdevice );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name — The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

type — The type of the element; one of:

- SND_MIXER_ETYPE_CAPTURE2
- SND_MIXER_ETYPE_PLAYBACK2

device — The PCM device that this element is a subdevice of; see *ado_mixer_element_pcm1()*.

subdevice — Not currently used; set it to 0.

Description:

This convenience function automates the creation of a PCM (type 2) element. A PCM type 2 element specifies the start of a playback subchannel. This element is used when a pcm1 supports multiple subchannels.

Returns:

A pointer to the newly allocated PCM (type 2) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), *ado_mixer_element_pcm1()*

ado_mixer_element_route_add()

Indicate an ordered relationship between two mixer elements

Synopsis:

```
#include <audio_driver.h>

int32_t ado_mixer_element_route_add
  ( ado_mixer_t *mixer,
    ado_mixer_delement_t *elem_before,
    ado_mixer_delement_t *elem );
```

Arguments:

mixer	A pointer to the **ado_mixer_t** structure for the mixer that the elements belong to.
elem_before	A pointer to the **ado_mixer_delement_t** structure for the mixer element that's the input to *elem*.
elem	A pointer to the **ado_mixer_delement_t** structure for the mixer element that gets its input from *elem_before*.

Description:

This function establishes an ordered relationship between two elements of the given mixer. The output from *elem_before* becomes the input to *elem*. When you use it repeatedly, this function builds route tables showing the relationships between all mixer elements.

Returns:

-1 if an error occurred (*errno* is set). Any other value indicates success.

Errors:

ENOMEM	Not enough free memory to extend the elements' route tables.

Classification:

QNX Neutrino

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

ado_mixer_element_sw1()

Create a switch (type 1) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_sw1 (
    ado_mixer_t *mixer,
    char *name,
    uint32_t number_of_switches,
    ado_mixer_delement_control_sw1_t *control,
    void *instance_data,
    void (*instance_free) (void *data) );
```

Arguments:

mixer
: A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name
: The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

number_of_switches
: The number of individual switches in the element. There's usually one switch for each voice of the stream passing through the element.

control
: A callback function, of type **ado_mixer_delement_control_sw1_t**, that sets the state of the switches.

  The prototype is:

```
int32_t control(
            MIXER_CONTEXT_T *context,
            ado_mixer_delement_t *element,
            uint8_t set,
            uint32_t *bitmap,
            void *instance_data );
```

instance_data A pointer to any instance data that you need to pass to the *control* callback. This can be a pointer to allocated memory, in which case you'll need to define the *instance_free* function.

If you need to access this instance data, you have to call *ado_mixer_get_element_instance_data()* because **`ado_mixer_delement_t`** is an opaque data type.

instance_free A function that must free any allocated instance data. It's called when the element is destroyed.

Description:

The *ado_mixer_element_sw1()* convenience function automates the creation of a switch (type 1) element. A switch type 1 element represents a switch with a single input and output that can be open or closed by a single control.

Returns:

A pointer to the newly allocated switch (type 1) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), ado_mixer_element_sw2(), ado_mixer_element_sw3(), ado_mixer_get_element_instance_data()

ado_mixer_element_sw2()

Create a switch (type 2) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_sw2
  ( ado_mixer_t *mixer,
    char *name,
    ado_mixer_delement_control_sw2_t *control,
    void *instance_data,
    void (*instance_free) (void *data) );
```

Arguments:

mixer
A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name
The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

control
A callback function, of type **ado_mixer_delement_control_sw2_t**, that sets the state of the switch.

The prototype is:

```
int32_t control(
            MIXER_CONTEXT_T *context,
            ado_mixer_delement_t *element,
            uint8_t set,
            uint32_t *val,
            void *instance_data );
```

instance_data
A pointer to any instance data that you need to pass to the *control* callback. This can be a pointer to allocated memory, in which case you'll need to define the *instance_free* function.

If you need to access this instance data, you have to call *ado_mixer_get_element_instance_data()* because **`ado_mixer_delement_t`** is an opaque data type.

instance_free A function that must free any allocated instance data. It's called when the element is destroyed.

Description:

The *ado_mixer_element_sw2()* convenience function automates the creation of a switch (type 2) element. A switch type 2 element represents a switch with a single input and output that can be open or closed with a separate control for each voice.

Returns:

A pointer to the newly allocated switch (type 2) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), ado_mixer_element_sw1(), ado_mixer_element_sw3(), ado_mixer_get_element_instance_data()

ado_mixer_element_sw3()

Create a switch (type 3) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_sw3
  ( ado_mixer_t *mixer,
    char *name,
    uint32_t type,
    uint32_t number_of_voices,
    snd_mixer_voice_t *voices,
    ado_mixer_delement_control_sw3_t *control,
    void *instance_data,
    void (*instance_free) (void *data) );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name — The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

type — The type of the switch; one of:

- SND_MIXER_SWITCH3_FULL_FEATURED
- SND_MIXER_SWITCH3_ALWAYS_DESTINATION
- SND_MIXER_SWITCH3_ALWAYS_ONE_DESTINATION
- SND_MIXER_SWITCH3_ONE_DESTINATION

number_of_voices

The number of voices passing through the element.

voices — An array of **snd_mixer_voice_t** structures. Each entry describes one of the voices.

For information about the **snd_mixer_voice_t** structure, see *ado_mixer_element_io()*.

control
A callback function, of type **ado_mixer_delement_control_sw3_t**, that controls the routing of the signals.

The prototype is:

```
int32_t control(
            MIXER_CONTEXT_T *context,
            ado_mixer_delement_t *element,
            uint8_t set,
            uint32_t *bitmap,
            void *instance_data );
```

instance_data
A pointer to any instance data that the *control* callback might need. This can be a pointer to allocated memory, in which case you'll need to define the *instance_free* function.

If you need to access this instance data, you have to call *ado_mixer_get_element_instance_data()* because **ado_mixer_delement_t** is an opaque data type.

instance_free
A function that must free any allocated instance data. It's called when the element is destroyed.

Description:

The *ado_mixer_element_sw3()* convenience function automates the creation of a switch (type 3) element. A switch type 3 element represents a switch with one or more inputs and outputs that can be routed to each other.

Returns:

A pointer to the newly allocated switch (type 3) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_create(), *ado_mixer_element_sw1()*, *ado_mixer_element_sw2()*, *ado_mixer_get_element_instance_data()*

ado_mixer_element_vol_range_max()

Read the maximum volume setting of the mixer volume element

Synopsis:

```
#include <audio_driver.h>

int32_t ado_mixer_element_vol_range_max(
            ado_mixer_delement_t *element );
```

Arguments:

element A pointer to the **ado_mixer_delement_t** structure for the mixer volume element.

Description:

The *ado_mixer_element_vol_range_max()* function reads the maximum volume setting of the mixer volume element *delement*.

Returns:

The maximum volume setting value.

Classification:

QNX Neutrino

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

See also:

ado_mixer_element_vol_range_min(), *ado_mixer_element_volume1()*

ado_mixer_element_vol_range_min()

Read the minimum volume setting of the mixer volume element

Synopsis:

```
#include <audio_driver.h>

int32_t ado_mixer_element_vol_range_min(
            ado_mixer_delement_t *delement );
```

Arguments:

delement A pointer to the **ado_mixer_delement_t** structure for the mixer volume element.

Description:

The *ado_mixer_element_vol_range_min()* function reads the minimum volume setting of the mixer volume element *delement*.

Returns:

The minimum volume setting value.

Classification:

QNX Neutrino

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

See also:

ado_mixer_element_vol_range_max(), *ado_mixer_element_volume1()*

ado_mixer_element_volume1()

Create a volume (type 1) element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_element_volume1
  ( ado_mixer_t *mixer,
    char *name,
    uint32_t number_of_voices,
    struct snd_mixer_element_volume1_range *ranges,
    ado_mixer_delement_control_volume1_t *control,
    void *instance_data,
    void (*instance_free) (void *data) );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name — The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

number_of_voices — The number of individual gain controls belonging to the element.

ranges — The range of adjustment for each gain control. The **snd_mixer_element_volume1_range** structure is defined as:

```
typedef struct snd_mixer_element_volume1_range
{
int32_t min, max;
int32_t min_dB, max_dB;
uint8_t   reserved[128];      /* must be filled with zero */
} snd_mixer_element_volume1_range_t;
```

control A callback function, of type **ado_mixer_delement_control_volume1_t**, that controls the gains of the analog signals passing through the element.

The prototype is:

```
int32_t control(
            MIXER_CONTEXT_T *context,
            ado_mixer_delement_t *element,
            uint8_t set,
            uint32_t *volumes,
            void *instance_data );
```

instance_data A pointer to any instance data that you need to pass to the *control* callback. This can be a pointer to allocated memory, in which case you'll need to define the *instance_free* function.

If you need to access this instance data, you have to call *ado_mixer_get_element_instance_data()* because **ado_mixer_delement_t** is an opaque data type.

instance_free A function that must free any allocated instance data. It's called when the element is destroyed.

Description:

The *ado_mixer_element_volume1()* convenience function automates the creation of a volume (type 1) element.

Returns:

A pointer to the newly allocated volume (type 1) element.

Classification:

QNX Neutrino

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

See also:

ado_mixer_element_vol_range_max(),
ado_mixer_element_vol_range_min(),
ado_mixer_get_element_instance_data()

ado_mixer_find_element()

Search a mixer for an element

Synopsis:

```
#include <audio_driver.h>

ado_mixer_delement_t *ado_mixer_find_element
  ( ado_mixer_t *mixer,
    int32_t type,
    int8_t *name,
    int32_t index );
```

Arguments:

mixer The mixer to search. This is a pointer to the **ado_mixer_t** structure created by *ado_mixer_create()*.

type The type of element; one of:

- SND_MIXER_ETYPE_INPUT
- SND_MIXER_ETYPE_ADC
- SND_MIXER_ETYPE_CAPTURE1
- SND_MIXER_ETYPE_CAPTURE2
- SND_MIXER_ETYPE_OUTPUT
- SND_MIXER_ETYPE_DAC
- SND_MIXER_ETYPE_PLAYBACK1
- SND_MIXER_ETYPE_PLAYBACK2
- SND_MIXER_ETYPE_SWITCH1
- SND_MIXER_ETYPE_SWITCH2
- SND_MIXER_ETYPE_SWITCH3
- SND_MIXER_ETYPE_VOLUME1
- SND_MIXER_ETYPE_VOLUME2
- SND_MIXER_ETYPE_ACCU1
- SND_MIXER_ETYPE_ACCU2
- SND_MIXER_ETYPE_ACCU3
- SND_MIXER_ETYPE_MUX1

- SND_MIXER_ETYPE_MUX2
- SND_MIXER_ETYPE_TONE_CONTROL1
- SND_MIXER_ETYPE_3D_EFFECT1
- SND_MIXER_ETYPE_EQUALIZER1
- SND_MIXER_ETYPE_PAN_CONTROL1
- SND_MIXER_ETYPE_PRE_EFFECT1

name The name of the element.

index The index of the element.

Description:

The *ado_mixer_find_element()* function searches all elements in a mixer for a given element.

Returns:

A pointer to the element, or NULL if no exact match was found.

Classification:

QNX Neutrino

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

See also:

ado_mixer_find_group()

ado_mixer_find_group()

Search a mixer for a group

Synopsis:

```
#include <audio_driver.h>

ado_mixer_dgroup_t *ado_mixer_find_group
  ( ado_mixer_t *mixer,
    int8_t *name,
    int32_t index );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure for the mixer.

name The name of the group.

index The index of the group.

Description:

The *ado_mixer_find_group()* searches in the given mixer for a group with the specified *name* and *index*.

Returns:

A pointer to the found group, or NULL if no exact match was found.

Classification:

QNX Neutrino

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

See also:

ado_mixer_find_element()

ado_mixer_get_context()

Get a pointer to a mixer's context structure

Synopsis:

```
#include <audio_driver.h>

MIXER_CONTEXT_T *ado_mixer_get_context (
                    ado_mixer_t *mixer );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure for the mixer.

Description:

The *ado_mixer_get_context()* function lets your Audio HW DLL access the *mixer*'s context structure. If you need to get the mixer's context, you have to use this function because **ado_mixer_t** is an opaque data type.

Returns:

A pointer to the mixer's context structure.

Classification:

QNX Neutrino

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

Access a mixer element's instance data

Synopsis:

```
#include <audio_driver.h>

void *ado_mixer_get_element_instance_data(
         ado_mixer_delement_t *element );
```

Arguments:

element	A pointer to the **ado_mixer_delement_t** structure for the mixer element.

Description:

The *ado_mixer_get_element_instance_data()* function gets a pointer to the instance data for the given mixer element. If you need to access this data, you'll need to use this function because **ado_mixer_delement_t** is an opaque data type. The library only passes the instance data around, preserving its connection to the element structure.

You can save instance data with the elements that you create by calling the following functions:

- *ado_mixer_element_accu3()*
- *ado_mixer_element_mux1()*
- *ado_mixer_element_mux2()*
- *ado_mixer_element_sw1()*
- *ado_mixer_element_sw2()*
- *ado_mixer_element_sw3()*
- *ado_mixer_element_volume1()*

Returns:

A pointer to the mixer element's instance data.

Classification:

QNX Neutrino

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

ado_mixer_lock()

Limit access to the mixer

Synopsis:

```
#include <audio_driver.h>

int32_t ado_mixer_lock( ado_mixer_t *mixer );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure for the mixer.

Description:

The *ado_mixer_lock()* locks the attribute structure, limiting access to the mixer. This function is a cover of *iofunc_attr_lock()*, using the mixer attributes.

The library automatically locks the mixer's attributes structure before any low-level mixer code is called, so you need this function only if some other functions, such as the PCM functions, need to manipulate the mixer structure.

Returns:

EOK	Success.
EAGAIN	On the first use, all kernel mutex objects were in use.

Classification:

QNX Neutrino

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

See also:

ado_mixer_unlock()

iofunc_attr_lock() in the QNX *Library Reference*

ado_mixer_playback_group_create()

Create a playback group

Synopsis:

```
#include <audio_driver.h>

ado_mixer_dgroup_t *ado_mixer_playback_group_create
  ( ado_mixer_t *mixer,
    char *name,
    uint32_t channels,
    ado_mixer_delement_t *vol_elem,
    ado_mixer_delement_t *mute_elem );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the group in. This structure was created by *ado_mixer_create()*.

name — The name of the group, which can be up to 31 characters long. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

channels — A bitmap of the channels in the group; any combination of:

- SND_MIXER_CHN_MASK_MONO
- SND_MIXER_CHN_MASK_FRONT_LEFT
- SND_MIXER_CHN_MASK_FRONT_RIGHT
- SND_MIXER_CHN_MASK_FRONT_CENTER
- SND_MIXER_CHN_MASK_REAR_LEFT
- SND_MIXER_CHN_MASK_REAR_RIGHT
- SND_MIXER_CHN_MASK_WOOFER
- SND_MIXER_CHN_MASK_STEREO

vol_elem — The volume element for the group.

mute_elem — The mute element for the group.

Description:

The *ado_mixer_playback_group_create()* function automates the allocation and filling of an **`ado_mixer_dgroup_t`** structure representing a channel in the playback direction.

Returns:

A pointer to the newly created playback group.

Classification:

QNX Neutrino

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

See also:

ado_mixer_capture_group_create()

ado_mixer_set_destroy_func()

Attach a function to the mixer to be called when the mixer is destroyed

Synopsis:

```
#include <audio_driver.h>

void ado_mixer_set_destroy_func
  ( ado_mixer_t *mixer,
    ado_mixer_destroy_t *destroy );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure for the mixer.

destroy A pointer to the cleanup function. Its prototype is:

```
int32_t destroy( MIXER_CONTEXT_T *context );
```

The **io-audio** manager ignores the value that the cleanup function returns.

Description:

The *ado_mixer_set_destroy_func()* attaches a function to the *mixer* that **io-audio** will call when the mixer is destroyed. You can use this function to do any required cleanup.

Classification:

QNX Neutrino

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

See also:

ado_mixer_set_reset_func()

ado_mixer_set_name()

Attach the character string name to the mixer

Synopsis:

```
#include <audio_driver.h>

void ado_mixer_set_name( ado_mixer_t *mixer,
                         char *name );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure for the mixer.

name The name to be attached to the mixer.

Description:

The *ado_mixer_set_name()* function attaches the given name to the given mixer.

A mixer has only one name. It's visible to programmers through the external API and should uniquely represent this specific mixer hardware.

Classification:

QNX Neutrino

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

ado_mixer_set_reset_func()

Attach a function to the mixer to be called when a hardware reset of the mixer occurs

Synopsis:

```
#include <audio_driver.h>

void ado_mixer_set_reset_func
  ( ado_mixer_t *mixer,
    ado_mixer_reset_t *reset );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure for the mixer.

reset A pointer to the reset function. Its prototype is:

```
int32_t reset( MIXER_CONTEXT_T *context );
```

The **io-audio** manager ignores the value that the reset function returns.

Description:

The *ado_mixer_set_reset_func()* attaches a function to the *mixer* that **io-audio** calls when a hardware reset of the mixer occurs.

Classification:

QNX Neutrino

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

See also:

ado_mixer_set_reset_func()

ado_mixer_switch_new()

Create a new mixer switch

Synopsis:

```
#include <audio_driver.h>

ado_dswitch_t *ado_mixer_switch_new
  ( ado_mixer_t *mixer,
    char *name,
    uint32_t type,
    uint32_t subtype,
    int32_t (*read) ( MIXER_CONTEXT_T *context,
                ado_dswitch_t *dswitch,
                snd_switch_t *cswitch,
                void *instance_data ),
    int32_t (*write) ( MIXER_CONTEXT_T *context,
                ado_dswitch_t *dswitch,
                snd_switch_t *cswitch,
                void *instance_data ),
    void *instance_data,
    void (*instance_free) (void *data) );
```

Arguments:

mixer — A pointer to the **ado_mixer_t** structure that specifies the mixer to create the element in. This structure was created by *ado_mixer_create()*.

name — The name of the element. Elements are referred to by name, so be careful; for some standard names, see **<asound.h>**.

type — The type of switch; one of:

- SND_SW_TYPE_BOOLEAN
- SND_SW_TYPE_BYTE
- SND_SW_TYPE_WORD
- SND_SW_TYPE_DWORD
- SND_SW_TYPE_LIST

subtype — The subtype of the switch; one of:

- SND_SW_SUBTYPE_DEC
- SND_SW_SUBTYPE_HEXA

read()	A callback that reads the state of the switch.
write()	A callback that writes the state of the switch.
instance_data	A pointer to any instance data that the *read* and *write()* callbacks might need. This can be a pointer to allocated memory, in which case you'll need to define the *instance_free* function.
instance_free	A function that must free any allocated instance data. It's called when the element is destroyed.

Description:

The *ado_mixer_switch_new()* function creates a new mixer switch.

A mixer switch isn't the same thing as a mixer switch element:

Mixer switch element	Used for things such as mutes and capture selects.
Mixer switch	Used less frequently, but still important to have available, e.g. PCM Loopback.

Returns:

A pointer to the newly allocated switch.

Classification:

QNX Neutrino

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

ado_mixer_unlock()

Unlock the attribute structure

Synopsis:

```
#include <audio_driver.h>

int32_t ado_mixer_unlock( ado_mixer_t *mixer );
```

Arguments:

mixer A pointer to the **ado_mixer_t** structure for the mixer.

Description:

The *ado_mixer_unlock()* function unlocks the attribute structure. This function is a cover of *iofunc_attr_unlock()* using the mixer attributes.

The library automatically locks the mixer's attributes structure before any low-level mixer code is called, so you need this function only if some other functions, such as the PCM functions, need to manipulate the mixer structure.

Returns:

EOK	Success.
EAGAIN	On the first use, all kernel mutex objects were in use.

Classification:

QNX Neutrino

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

See also:

ado_mixer_lock()

iofunc_attr_unlock() in the QNX *Library Reference*

ado_mutex_destroy()

Destroy a mutex

Synopsis:

```
#include <audio_driver.h>

#define ado_mutex_destroy( mutex );
```

Arguments:

mutex The mutex to destroy. The mutex is of type **`pthread_mutex_t *`**.

Description:

The *ado_mutex_destroy()* macro destroys the given mutex.

This macro is defined as *pthread_mutex_destroy()*, or *ado_mutex_destroy_debug()* if ADO_MUTEX_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message to help you locate a mutex problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_mutex_destroy()*:

EOK	Success.
EBUSY	The *mutex* is locked.
EINVAL	The given mutex is invalid.

Classification:

QNX Neutrino

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

See also:

ado_mutex_init(), *ado_mutex_lock()*, *ado_mutex_unlock()*

pthread_mutex_destroy() in the QNX *Library Reference*

ado_mutex_init()

Initialize a mutex

Synopsis:

```
#include <audio_driver.h>

#define ado_mutex_init( mutex )
```

Arguments:

mutex The mutex to initialize. The mutex is of type **`pthread_mutex_t *`**.

Description:

The *ado_mutex_init()* macro initializes the given mutex.

This macro is defined as *pthread_mutex_init()* (passing NULL for the *attr* argument), or *ado_mutex_init_debug()* if ADO_MUTEX_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message to help you locate a mutex problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_mutex_init()*:

EOK	Success.
EAGAIN	All kernel synchronization objects are in use.
EBUSY	Previously initialized but undestroyed mutex *mutex*.
EFAULT	A fault occurred when the kernel tried to access *mutex* or *attr*.
EINVAL	The value specified by *attr* is invalid.

Classification:

QNX Neutrino

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

See also:

ado_mutex_destroy(), *ado_mutex_lock()*, *ado_mutex_unlock()*

pthread_mutex_init() in the QNX *Library Reference*

ado_mutex_lock()

Lock a mutex

Synopsis:

```
#include <audio_driver.h>

#define ado_mutex_lock( mutex )
```

Arguments:

mutex The mutex to lock. The mutex is of type **`pthread_mutex_t *`**.

Description:

The *ado_mutex_lock()* macro locks the given mutex.

This macro is defined as *pthread_mutex_lock()*, or *ado_mutex_lock_debug()* if ADO_MUTEX_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message so as to aid in locating a mutex problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_mutex_lock()*:

EOK	Success.
EAGAIN	Insufficient system resources available to lock the mutex.
EDEADLK	The calling thread already owns *mutex*, and the mutex doesn't allow recursive behavior.
EINVAL	The given mutex is invalid.

Classification:

QNX Neutrino

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

See also:

ado_mutex_destroy(), *ado_mutex_init()*, *ado_mutex_unlock()*

pthread_mutex_lock() in the QNX *Library Reference*

ado_mutex_unlock()

Unlock a mutex

Synopsis:

```
#include <audio_driver.h>

#define ado_mutex_unlock( mutex )
```

Arguments:

mutex The mutex to unlock. The mutex is of type **`pthread_mutex_t *`**.

Description:

The *ado_mutex_unlock()* macro unlocks the given mutex.

This macro is defined as *pthread_mutex_unlock()*, or *ado_mutex_unlock_debug()* if ADO_MUTEX_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message to help you locate a mutex problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_mutex_unlock()*:

EOK	Success.
EINVAL	The given mutex is invalid.
EPERM	The current thread doesn't own *mutex*.

Classification:

QNX Neutrino

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

See also:

ado_mutex_destroy(), *ado_mutex_init()*, *ado_mutex_lock()*

pthread_mutex_unlock() in the QNX *Library Reference*

ado_pci

Data structure that describes a selected PCI card

Synopsis:

```
struct ado_pci
{
    int32_t     index;
    int32_t     id;
    void *      handle;
    uint16_t    vendor;
    uint16_t    device;
    uint16_t    subsystem;
    uint16_t    subsysvendor;
    uint8_t     devfunc;
    uint8_t     revision;
    uint8_t     class_protocol;
    uint8_t     zero0;
    uint32_t    irq;
    uint64_t    bmstr;
    uint64_t    iobase[6];
    uint32_t    iolen[6];
    uint32_t    class;
    uint32_t    spare[3];
};
```

Description:

The **ado_pci** structure describes the PCI card selected by *ado_pci_device()*. The members include:

index	Index of the device
id	The connection handle to the PCI server. See *pci_attach()* in the QNX *Library Reference*.
handle	A handle that you can use to identify the PCI device. See *pci_attach_device()* in the QNX *Library Reference*.
vendor	Vendor ID.
device	Device ID.

subsystem	Subsystem ID.
subsysvendor	Subsystem vendor ID.
devfunc	Device/function ID.
revision	Device revision
class_protocol	Class protocol.
irq	Interrupt number.
bmstr	Translation from the CPU busmaster address to the PCI busmaster address.
iobase[6]	CPU base address.
iolen[6]	Size of the base address aperture into the board.
class	Class code.

Classification:

QNX Neutrino

See also:

ado_pci_device(), *ado_pci_release()*

pci_attach(), *pci_attach_device()* in the QNX *Library Reference*

ado_pci_device()

Try to connect to a specified PCI card

Synopsis:

```
#include <audio_driver.h>

struct ado_pci *ado_pci_device( int32_t vendor,
                                int32_t device,
                                char *options );
```

Arguments:

vendor	The vendor ID.
device	The device ID.
options	The command-line arguments passed to your Audio HW DLL as the *args* argument to *ctrl_init()* (see the Organization of a Driver chapter).

Description:

This function tries to connect to a PCI card that matches the given vendor and device IDs.

Returns:

A pointer to a **ado_pci** structure that describes the PCI card, or NULL if an error occurred (e.g. there was no card to attach to).

Classification:

QNX Neutrino

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

See also:

ado_pci

ado_pci_release()

Detach from a given PCI card

Synopsis:

```
void ado_pci_release( struct ado_pci *pci );
```

Arguments:

pci A pointer to the **ado_pci** structure that describes the PCI card. This is the pointer returned by *ado_pci_device()*.

Description:

The *ado_pci_release()* function detaches from the PCI card described by the **ado_pci** structure pointed to by *pci*. The *ado_pci_release()* function disconnects from the PCI server and frees the **ado_pci** structure.

Classification:

QNX Neutrino

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

See also:

ado_pci, *ado_pci_device()*

ado_pcm_cap_t

Data structure of capabilities of a PCM device

Synopsis:

```
struct ado_pcm_cap
{
   uint32_t chn_flags;
   uint32_t formats;
   uint32_t rates;
   uint32_t min_rate;
   uint32_t max_rate;
   uint32_t min_voices;
   uint32_t max_voices;
   uint32_t min_fragsize;
   uint32_t max_fragsize;
   uint32_t max_dma_size;
   uint32_t max_frags;
} ado_pcm_cap_t;
```

Description:

The **ado_pcm_cap_t** structure describes the capabilities of the playback or capture portion of a PCM device. The members include:

chn_flags — One of the SND_PCM_CHNINFO_* flags. For more information, see *snd_pcm_channel_info()* in the QNX *Audio Developer's Guide*.

formats — The formats supported; any combination of the SND_PCM_FMT_* flags. For more information, see *snd_pcm_find()* in the QNX *Audio Developer's Guide*.

rates — The sample rate; a combination of:

- SND_PCM_RATE_8000
- SND_PCM_RATE_11025
- SND_PCM_RATE_16000
- SND_PCM_RATE_22050
- SND_PCM_RATE_32000

- SND_PCM_RATE_44100
- SND_PCM_RATE_48000
- SND_PCM_RATE_88200
- SND_PCM_RATE_96000
- SND_PCM_RATE_176400
- SND_PCM_RATE_192000

or one of:

- SND_PCM_RATE_CONTINUOUS — see *min_rate* and *max_rate*.
- SND_PCM_RATE_8000_44100 — all the above rates from 8000 through 44100.
- SND_PCM_RATE_8000_48000 — all the above rates from 8000 through 48000.

min_rate	The minimum sample rate if *rates* is SND_PCM_RATE_CONTINUOUS.
max_rate	The maximum sample rate if *rates* is SND_PCM_RATE_CONTINUOUS.
min_voices	Minimum number of voices.
max_voices	Maximum number of voices.
min_fragsize	Minimum fragment size for DMA transfer.
max_fragsize	Maximum fragment size for DMA transfer.
max_dma_size	Maximum DMA buffer size. A value of 0 means no limit.
max_frags	Maximum number of fragments. A value of 0 means no limit.

Classification:

QNX Neutrino

See also:

ado_pcm_create()

snd_pcm_channel_info(), *snd_pcm_find()* in the QNX *Audio Developer's Guide*

ado_pcm_config_t

Data structure that describes the configuration of a PCM subchannel

Synopsis:

```
struct  ado_pcm_config  {
    snd_pcm_format_t            format;
    union
    {
        struct
        {
            int32_t         frag_size;
            int32_t         frags_min;
            int32_t         frags_max;
            int32_t         frags_total;
            ado_pcm_mmap_t  mmap;
        } block;
    } mode;
    ado_pcm_dmabuf_t        dmabuf;
    int32_t                 mixer_device;   /* mixer device */
    snd_mixer_eid_t         mixer_eid;      /* pcm subchn source element
    snd_mixer_gid_t         mixer_gid;      /* lowest level mixer group s
};
```

Description:

This structure is passed to the *aquire*, *prepare*, and *trigger* device PCM callback functions, and defines how the PCM subchannel is to be configured. For more information about these callbacks, see **ado_pcm_hw_t**.

The members are:

format The format that the subchannel must support, including the rate, bits, endianness, and channels. The **snd_pcm_format_t** is defined as:

```
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];
```

```
} snd_pcm_format_t;
```

mode This union has parameters for block and stream mode, but stream mode is deprecated. The *block* structure includes:

- *frag_size* — the fragment size that the client is requesting.
- *frags_min*, *frags_max* — the minimum and maximum fragment sizes that the client wants.
- *frags_total* — the total number of fragments in the DMA buffer.

In most cases, the DMA buffer allocated in the *aquire* callback is *frags_total* × *frag_size* in length. But if the hardware has special requirements such that *frag_size* must be changed, the other members are there to help you select the best buffer size.

dmabuf A structure the defines the DMA buffer's virtual address, physical address, size, and name if it's to be shared. The **ado_pcm_dmabuf** structure is defined as:

```
struct  ado_pcm_dmabuf  {
        uint8_t         *addr;
        off64_t         phys_addr;
        size_t          size;
        int8_t          name[QNX_SHM_NAME_LEN];
};
```

mixer_device, *mixer_eid*, *mixer_gid*

The mixer members give the best associated mixer, element and group to do with this PCM device.

Classification:

QNX Neutrino

See also:

ado_pcm_hw_t

ado_pcm_chn_mixer()

Logically associate a mixer element and group with a PCM device

Synopsis:

```
#include <audio_driver.h>

void ado_pcm_chn_mixer(
  ado_pcm_t *pcm,
  enum pcm_chn_type type,
  ado_mixer_t *mixer,
  ado_mixer_delement_t *delement,
  ado_mixer_dgroup_t *dgroup );
```

Arguments:

pcm	A pointer to the **ado_pcm_t** structure created for the PCM device when you called *ado_pcm_create()*.
type	The type of channel; one of: • ADO_PCM_CHANNEL_CAPTURE • ADO_PCM_CHANNEL_PLAYBACK
mixer	The mixer that contains both the element and the group.
delement	The mixer PCM element to be associated with the PCM channel.
dgroup	The mixer group that best controls the PCM channel.

Description:

The *ado_pcm_chn_mixer()* function logically associates a mixer element and group with the specified PCM device for the given PCM channel type.

Classification:

QNX Neutrino

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

See also:

ado_pcm_create()

ado_pcm_create()

Create a PCM audio device

Synopsis:

```
#include <audio_driver.h>

int32_t ado_pcm_create (
    ado_card_t *card,
    char *name,
    uint32_t flags,
    char *id,
    uint32_t play_subchns,
    ado_pcm_cap_t *play_cap,
    ado_pcm_hw_t *play_hw,
    uint32_t cap_subchns,
    ado_pcm_cap_t *cap_cap,
    ado_pcm_hw_t *cap_hw,
    ado_pcm_t **rpcm );
```

Arguments:

card — The *card* argument that **io-audio** passed to your Audio HW DLL's *ctrl_init()* function (see the Organization of a Driver chapter).

name — The name of the new device. This is usually a variation of the card name, and is used only for information display by client applications.

flags — Capability flags for the device; one of the SND_PCM_INFO_* flags defined in **asound.h**. For more information, see *snd_pcm_info()* in the QNX *Audio Developer's Guide*.

These flags are used to identify device capabilities only.

id — The name of the new device.

play_subchns — The maximum number of simultaneous playback subchannels that the new device can support.

play_cap	A pointer to a static **ado_pcm_cap_t** structure that describes the static capabilities of the playback portion of the device. These are the capabilities that the first subchannel opened could use. As more subchannels are opened, the device capabilities may decrease and become unavailable.
play_hw	A pointer to a **ado_pcm_hw_t** structure that specifies all of the callbacks for the playback portion of the PCM device. The **io-audio** manager invokes these callbacks when something needs to be done with respect to the PCM device.
cap_subchns	The maximum number of simultaneous capture subchannels that the new device can support.
cap_cap	A pointer to a static **ado_pcm_cap_t** structure that describes the static capabilities of the capture portion of the device. These are the capabilities that the first subchannel opened could use. As more subchannels are opened, the device capabilities may decrease and become unavailable.
cap_hw	A pointer to a **ado_pcm_hw_t** structure that specifies all of the callbacks for the capture portion of the PCM device. The **io-audio** manager invokes these callbacks when something needs to be done with respect to the PCM device.
rpcm	A pointer to a memory location where *ado_pcm_create()* can store a pointer to the new PCM device. You'll need this address for additional API calls.

Description:

The *ado_pcm_create()* function creates a PCM audio device and attaches it to the given card.

Returns:

Zero on success, or **-1** on failure (*errno* is set).

Errors:

ENOMEM	Memory couldn't be allocated to hold the device structures.
EINVAL	One or more of the function's arguments are invalid.

Classification:

QNX Neutrino

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

See also:

ado_pcm_cap_t, **ado_pcm_hw_t**

ctrl_init() in the Organization of a Driver chapter, Handling PCM Audio Data chapter

snd_pcm_info() in the QNX *Audio Developer's Guide*

ado_pcm_dma_int_size()

Obtain the fragment size of a PCM channel

Synopsis:

```
#include <audio_driver.h>

uint32_t ado_pcm_dma_int_size (
            ado_pcm_config_t *config );
```

Arguments:

config A pointer to the **ado_pcm_config_t** that describes the PCM channel.

Description:

The *ado_pcm_dma_int_size()* function returns the fragment size of the PCM channel given by the config structure *config*. This fragment size is the number of bytes to be transferred via DMA to or from the DMA buffer before an interrupt is signalled.

Returns:

The fragment size for the PCM DMA transfer.

Classification:

QNX Neutrino

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

ado_pcm_format_bit_width()

Obtain the sample width, in bits, for a given format

Synopsis:

```
#include <audio_driver.h>

size_t ado_pcm_format_bit_width( int format );
```

Arguments:

format The format whose sample width you need to determine.

Description:

The *ado_pcm_format_bit_width()* function returns the sample width in bits for the given *format*. For example, **SND_PCM_FMT_S16** returns 16.

Returns:

The bit width of the given format.

Classification:

QNX Neutrino

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

ado_pcm_hw_t

Data structure of callbacks for PCM devices

Synopsis:

```
struct ado_pcm_hw
{
int32_t (*capabilities) (HW_CONTEXT_T *hw_context,
                         snd_pcm_channel_info_t *info );

int32_t (*aquire) (HW_CONTEXT_T *hw_context,
                   PCM_SUBCHN_CONTEXT_T **PCM_SUBCHN_CONTEXT,
                   ado_pcm_config_t *config,
                   ado_pcm_subchn_t *subchn,
                   uint32_t *why_failed );

int32_t (*release) (HW_CONTEXT_T *hw_context,
                   PCM_SUBCHN_CONTEXT_T *PCM_SUBCHN_CONTEXT,
                   ado_pcm_config_t *config);

int32_t (*prepare) (HW_CONTEXT_T *hw_context,
                   PCM_SUBCHN_CONTEXT_T *PCM_SUBCHN_CONTEXT,
                   ado_pcm_config_t *config);

int32_t (*trigger) (HW_CONTEXT_T *hw_context,
                   PCM_SUBCHN_CONTEXT_T *PCM_SUBCHN_CONTEXT,
                   uint32_t cmd);

uint32_t (*position) (HW_CONTEXT_T *hw_context,
                     PCM_SUBCHN_CONTEXT_T *PCM_SUBCHN_CONTEXT,
                     ado_pcm_config_t *config);

int32_t (*reconstitute) (ado_pcm_config_t * config, int8_t *dmaptr,
                         size_t size);
} ado_pcm_hw_t;
```

Description:

The **ado_pcm_hw_t** structure specifies the callbacks that you must provide for your PCM device's playback and/or capture portions.

aquire

The prototype is:

```
int32_t (*aquire)( HW_CONTEXT_T *hw_context,
                   PCM_SUBCHN_CONTEXT_T **PCM_SUBCHN_CONTEXT,
                   ado_pcm_config_t *config,
                   ado_pcm_subchn_t *subchn,
                   uint32_t *why_failed );
```

This is the function that the upper driver layers call when a client attempts to open a PCM stream.

The name of this member is spelt incorrectly; it should be "acquire" in the declaration of the structure. Unfortunately, it has to remain misspelt to be compatible with drivers that you and other developers may have already written. :-(

The arguments are:

hw_context A pointer to the context structure set up in the *ctrl_init()* function (see the Organization of a Driver chapter). You need it so that you can access the hardware registers.

PCM_SUBCHN_CONTEXT

A pointer that you can set to any structure for later reference by the other callbacks; you can set the type by defining **PCM_SUBCHN_CONTEXT_T**.

config A pointer to a **ado_pcm_config_t** structure that contains all the parameters about how the channel is to be set up.

subchn A pointer to the **ado_pcm_subchn_t** structure for the subchannel. You need to pass this pointer various other functions.

why_failed A pointer to a variable that you can use to return a more detailed reason as to why establishing the channel failed. You should set this variable to 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

This function is called only when the client asks for a setup that's within the capabilities of the device. This is done by examining the capabilities structure that was passed in as part of the PCM device creation.

The main responsibility of this call is to verify that the hardware can accommodate this request, given its current state. Then the callback must allocate any hardware necessary to fulfill the request, and allocate the DMA buffer for the channel. The important idea here is that on a card that supports multiple subchannels, there may be a finite amount of resources to accommodate user requests. So if a request is received when all required resources are being used, the request has to fail even though fewer than the total subchannels are active.

A very good real-world example is a card with 8 sample rate converters that supports 24 streams simultaneously at a native sample rate of 48 kHz. If 9 clients attempt to play 22 kHz data, one fails, but up to 16 additional requests at 48 kHz pass.

This callback should return:

EOK	Success.
ENOMEM	Not enough memory.
EAGAIN	No channel was available; set *why_failed* accordingly.

release

The prototype is:

```
int32_t (*release)( HW_CONTEXT_T *hw_context,
                    PCM_SUBCHN_CONTEXT_T *PCM_SUBCHN_CONTEXT,
                    ado_pcm_config_t *config);
```

The upper layers of the driver call this function when the client application closes its connection to the device. This function is the reciprocal of the *aquire* callback; it must free any acquired hardware and release the memory from the DMA buffer.

This function should return EOK.

prepare

The prototype is:

```
int32_t (*prepare)( HW_CONTEXT_T *hw_context,
                    PCM_SUBCHN_CONTEXT_T *PCM_SUBCHN_CONTEXT,
                    ado_pcm_config_t *config);
```

The upper layers of the driver call this function to prepare the hardware before it's started up. The primary reason this function is here is that the stream may stop and then restart at any time because of an underrun, so the hardware must be reset. Usually this function simply resets the DMA engine to start from the beginning of the DMA buffer.

This function should return EOK.

trigger

The prototype is:

```
int32_t (*trigger)( HW_CONTEXT_T *hw_context,
                    PCM_SUBCHN_CONTEXT_T *PCM_SUBCHN_CONTEXT,
                    uint32_t cmd);
```

This function is the hardware start and stop callback. The *cmd* parameter states the go or stop condition, and is one of:

- ADO_PCM_TRIGGER_GO
- ADO_PCM_TRIGGER_STOP
- ADO_PCM_TRIGGER_SYNC_GO

This function should return EOK.

position

The prototype is:

```
uint32_t (*position)( HW_CONTEXT_T *hw_context,
                      PCM_SUBCHN_CONTEXT_T *PCM_SUBCHN_CONTEXT,
                      ado_pcm_config_t *config);
```

This function must return where the DMA engine is within the current fragment. It's used by the upper layer of the driver to return timing information to the client.

This function should return the number of bytes played from the beginning of the fragment.

reconstitute

The prototype is:

```
int32_t (*reconstitute)( ado_pcm_config_t *config,
                         int8_t *dmaptr,
                         size_t size);
```

This function is a catchall for hardware with very strange format support. It's used to reformat the data in the DMA buffer for the strange hardware requirements.

This callback has been used only once to date for hardware that supported 20 bits of resolution in a 32-bit sample. The upper layers of the driver filled the buffer with the MSB of the sample at bit position 31. The hardware wanted the MSB to be at bit position 19, so the *reconstitute* function performed a 12-bit shift.

This function should return EOK.

capabilities

The prototype is:

```
int32_t (*capabilities)( HW_CONTEXT_T *hw_context,
                         snd_pcm_channel_info_t *info );
```

This function is used to return to the client the capabilities of the device at this instant. When the device was created, its static capabilities were passed in as an argument; however, if a number of subchannels are already running, the device may no longer have the ability to support those capabilities.

The upper driver layers call this function after copying the static capabilities into the *info* structure. This function should simply remove some options from the *info* structure, based on what hardware is currently not allocated. In the most extreme case where the device only supports one subchannel, and it's already in use, the function should remove all capabilities from the *info* structure.

You can call *ado_pcm_subchn_is_channel()* to determine the type of subchannel. By doing this, you can use the same *capabilities* callback for capture and playback subchannels.

For details about the **snd_pcm_channel_info_t** structure, see *snd_pcm_channel_info()* in the QNX *Audio Developer's Guide*.

The *capabilities* function should return EOK.

Classification:

QNX Neutrino

See also:

ado_pcm_config_t, *ado_pcm_create()*

snd_pcm_channel_info() in the QNX Neutrino *Audio Developer's Guide*

ado_pcm_subchn_caps()

Get a pointer to the capabilities structure for a subchannel

Synopsis:

```
#include <audio_driver.h>

ado_pcm_cap_t *ado_pcm_subchn_caps (
                    ado_pcm_subchn_t *subchn );
```

Arguments:

subchn A pointer to the **ado_pcm_subchn_t** structure that describes the subchannel.

Description:

The *ado_pcm_subchn_caps()* function returns a pointer to the capabilities structure for the *subchannel*. This function is necessary because the **ado_pcm_subchn_t** structure is a opaque data type.

Returns:

A pointer to a **ado_pcm_cap_t** structure that describes the capabilities for the *subchn*, or NULL on error.

Classification:

QNX Neutrino

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

See also:

`ado_pcm_cap_t`

ado_pcm_subchn_is_channel()

Check if a channel is a subchannel of a PCM device

Synopsis:

```
#include <audio_driver.h>

int32_t ado_pcm_subchn_is_channel
  ( ado_pcm_subchn_t *subchn,
    ado_pcm_t *pcm,
    enum pcm_chn_type chn_type );
```

Arguments:

subchn A pointer to the **ado_pcm_subchn_t** structure for the subchannel.

pcm A pointer to the **ado_pcm_t** structure created for the PCM device when you called *ado_pcm_create()*.

chn_type The channel type; one of:

- ADO_PCM_CHANNEL_CAPTURE
- ADO_PCM_CHANNEL_PLAYBACK

Description:

The *ado_pcm_subchn_is_channel()* function is a convenience function for checking if the *subchn* is a subchannel of the a PCM device. This function is necessary because the **subchn** structure is a opaque internal structure. This function is handy in the PCM *capabilities()* callback (see **ado_pcm_hw_t**) because all the devices can share the same callback, which uses this function to tell them apart.

Returns:

True if *subchn* is a subchannel of the PCM device *pcm* channel of type *type*.

Classification:

QNX Neutrino

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

See also:

`ado_pcm_hw_t`

ado_pcm_subchn_mixer()

Logically associate a mixer element and group with a PCM subchannel device

Synopsis:

```
#include <audio_driver.h>

void ado_pcm_subchn_mixer
  ( ado_pcm_subchn_t *subchn,
    ado_mixer_t *mixer,
    ado_mixer_delement_t *delement,
    ado_mixer_dgroup_t *dgroup );
```

Arguments:

subchn	The PCM subchannel device.
mixer	The mixer that contains both the element and the group.
delement	The mixer element associated with the subchannel.
dgroup	The mixer group that best controls the subchannel.

Description:

The *ado_pcm_subchn_mixer()* function logically associates a mixer element and group with a PCM subchannel device.

Classification:

QNX Neutrino

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

ado_pcm_subchn_mixer_create()

Create a PCM subchannel mixer

Synopsis:

```
#include <audio_driver.h>

ado_pcm_subchn_mixer_t *ado_pcm_subchn_mixer_create (
    ado_pcm_subchn_t *subchn,
    ado_mixer_t *mixer,
    ado_pcm_subchn_mixer_config_t *config );
```

Arguments:

subchn	The subchannel to associate with.
mixer	The mixer on which to attach the new controls.
config	A pointer to a **ado_pcm_subchn_mixer_config_t** structure (see below) that describes the type of the controls the hardware has.

Description:

The *ado_pcm_subchn_mixer_create()* function is a convenience function for creating mixer elements and callbacks for volume control of a PCM subchannel device. If your audio chip supports more than one simultaneous subchannel, and has gain and or mute controls for each subchannel, this convenience function will simplify their control.

ado_pcm_subchn_mixer_config_t structure

```
typedef struct ado_pcm_subchn_mixer_config
{
HW_CONTEXT_T  *hw_context;

PCM_SUBCHN_CONTEXT_T  *pcm_sc_context;

uint32_t  channel_mask;

uint32_t  volume_jointly:1;

snd_mixer_element_volume1_range_t  volume_range;

uint32_t  mute_jointly:1;
```

```
void (*volume_set)  (HW_CONTEXT_T * hw_context,
                     PCM_SUBCHN_CONTEXT_T * pcm_sc_context, int32_t * volumes,
                     int32_t mute, ado_pcm_subchn_mixer_config_t *config);

void (*mute_set)  (HW_CONTEXT_T * hw_context,
                   PCM_SUBCHN_CONTEXT_T * pcm_sc_context,  int32_t * volumes,
                   int32_t mute, ado_pcm_subchn_mixer_config_t *config);
} ado_pcm_subchn_mixer_config_t;
```

channel_mask — The number and position of voices, and the new mixer controls. Most commonly this will be 11b or 0x3h for the front left and right channels.

volume_range — The gain range for the volume element. The range is assumed to be from least to most gain.

volume_set — An optional callback function used to set the volume gain. When called by **io-audio** the volumes pointer is an array, *number_of_voices* deep, with the level of gain to set on the subchannel. The level is guaranteed to be within the range specified. If this argument is NULL, no volume control is created.

mute_set — An optional callback function used to mute the subchannel output. When called, the *mute* argument uses a bit for each channel to indicate the mute state. If the bit is on, the subchannel is to be muted; if the bit is zero, the subchannel is to be unmuted. If this argument is NULL, no mute control is created.

Returns:

A pointer to the new *subchn* mixer, or NULL if an error occurred (*errno* is set).

The **ado_pcm_subchn_mixer_t** structure is an opaque data type. You'll need a pointer to it when you call *ado_pcm_subchn_mixer_destroy()* to destroy the subchannel mixer.

Examples:

This example of how to setup a subchannel mixer is taken from the vortex driver. Due to legal restrictions, the full source to the vortex driver isn't availiable in the DDK. The vortex has individual volume and mute controls for each channel of the stereo stream. The volume controls have a range of 0 to 255 settings, corresponding to a decibel range of -102.35 dB to 0 dB.

The vortex hardware functions, *vortex_mixer_input_gain()* and *vortex_mixer_input_mute()* set the hardware volume and mute. But in order to work, those functions need to know which hardware mixer and which mixer input the signals are routed through, so this information is kept in the **PCM_SUBCHN_CONTEXT_T *vsc** variable.

```
static void
vortex_subchn_volume_set( HW_CONTEXT_T * vortex,
   PCM_SUBCHN_CONTEXT_T * vsc, int32_t * volumes,
   int32_t mute, ado_pcm_subchn_mixer_config_t * config)
{
    if (vortex->card_type == VORTEX_CARD_TYPE_AU8830)
    {
        vortex_mixer_input_gain_8830 (vortex, vsc->mixL,
           vsc->mixinL, volumes[0]);
        vortex_mixer_input_gain_8830 (vortex, vsc->mixR,
           vsc->mixinR, volumes[1]);
    }
    else
    {
        vortex_mixer_input_gain_8820 (vortex, vsc->mixL,
           vsc->mixinL, volumes[0]);
        vortex_mixer_input_gain_8820 (vortex, vsc->mixR,
           vsc->mixinR, volumes[1]);
    }
}

static void
vortex_subchn_mute_set (HW_CONTEXT_T * vortex,
   PCM_SUBCHN_CONTEXT_T * vsc, int32_t * volumes,
```

```
   int32_t mute, ado_pcm_subchn_mixer_config_t * config)
{
    if (vortex->card_type == VORTEX_CARD_TYPE_AU8830)
    {
        vortex_mixer_input_mute_8830 (vortex, vsc->mixL,
           vsc->mixinL, mute & (1 << 0));
        vortex_mixer_input_mute_8830 (vortex, vsc->mixR,
           vsc->mixinR, mute & (1 << 1));
    }
    else
    {
        vortex_mixer_input_mute_8820 (vortex, vsc->mixL,
           vsc->mixinL, mute & (1 << 0));
        vortex_mixer_input_mute_8820 (vortex, vsc->mixR,
           vsc->mixinR, mute & (1 << 1));
    }
}


int32_t
vortex_playback_aquire (HW_CONTEXT_T * vortex,
   PCM_SUBCHN_CONTEXT_T ** vsc,
   ado_pcm_config_t * config, ado_pcm_subchn_t * subchn,
   uint32_t * why_failed)
{
    .
    .
    .
    memset (&vsc_mix_cfg, 0, sizeof (vsc_mix_cfg));
    vsc_mix_cfg.hw_context = vortex;
    vsc_mix_cfg.pcm_sc_context = *vsc;
    vsc_mix_cfg.channel_mask = SND_MIXER_CHN_MASK_STEREO;
    vsc_mix_cfg.volume_range.min = 0;
    vsc_mix_cfg.volume_range.max = 0xff;
    vsc_mix_cfg.volume_range.min_dB = -10235;
    vsc_mix_cfg.volume_range.max_dB = 0;
    vsc_mix_cfg.volume_set = vortex_subchn_volume_set;
    vsc_mix_cfg.mute_set = vortex_subchn_mute_set;
    if (((*vsc)->scmix = ado_pcm_subchn_mixer_create (
       subchn, vortex->mixer, &vsc_mix_cfg)) == NULL)
    {
        return (ENOMEM);
    }
    .
    .
    .
}

int32_t
vortex_playback_release (HW_CONTEXT_T * vortex,
    PCM_SUBCHN_CONTEXT_T * vsc,
    ado_pcm_config_t * config)
{
```

```
    ⋮
    ado_pcm_subchn_mixer_destroy (vsc->scmix);
    ⋮
}
```

Classification:

QNX Neutrino

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

See also:

ado_pcm_subchn_mixer_destroy()

ado_pcm_subchn_mixer_destroy()

Destroy a PCM subchannel mixer

Synopsis:

```
#include <audio_driver.h>

void ado_pcm_subchn_mixer_destroy(
        ado_pcm_subchn_mixer_t *sc_mixer );
```

Arguments:

sc_mixer A pointer to a **ado_pcm_subchn_mixer_t** structure that describes the PCM subchannel mixer. This is the pointer returned by *ado_pcm_subchn_mixer_create()*.

Description:

The *ado_pcm_subchn_mixer_destroy()* function frees all memory, and then destroys the mixer elements and structures associated with the given subchannel mixer.

Classification:

QNX Neutrino

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

See also:

ado_pcm_subchn_mixer_create()

ado_pcm_sw_mix()

Provide a mechanism for an audio chip to support multiple simultaneous streams

Synopsis:

```
#include <audio_driver.h>

int32_t ado_pcm_sw_mix ( ado_card_t *card,
                         ado_pcm_t *pcm,
                         ado_mixer_t *mixer );
```

Arguments:

card The *card* argument that **io-audio** passed to your Audio HW DLL's *ctrl_init()* function (see the Organization of a Driver chapter).

pcm The PCM device the software mixer is built on. This is a pointer to the **ado_pcm_t** structure created for the PCM device when you called *ado_pcm_create()*.

mixer The mixer in which to create the subchannel mixer groups.

Description:

The *ado_pcm_sw_mix()* function provides a mechanism whereby an audio chip that supports only one hardware PCM subchannel can support multiple simultaneous streams.

The number of streams supported is a function of the **io-audio** architecture and your driver can't change it.

This function uses CPU power to mix the multiple streams together into one stream that it then sends to the PCM device.

The software mixer is implemented by creating a new PCM device attached to the card and creating the mixer controls for the new subchannel in the *mixer* specified.

The hardware subchannel is acquired only when the software mixer needs it, so it's possible for applications to open the hardware device directly, although this then prevents the software mixer from accepting a stream because the real hardware device is already in use.

Currently, software mixing is supported only on the playback channel of the PCM device.

Returns:

Zero on success, or -1 if an error occurred.

Classification:

QNX Neutrino

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

ado_realloc()

Allocate, reallocate, or free a block of memory

Synopsis:

```
#include <audio_driver.h>

void *ado_realloc( void *ptr,
                   size_t size);
```

Arguments:

ptr A pointer to the block of memory.

size The new size of the block, in bytes.

Description:

The *ado_realloc()* macro allocates, reallocates, or frees a block of memory.

This macro is defined as *realloc()*, or *ado_realloc_debug()* if ADO_DEBUG is defined; see "Debugging an audio driver" in the Organization of a Driver chapter.

The advantage of using the debug variant is that it tracks the memory allocated until it's freed; see *ado_memory_dump()*.

Returns:

Same as *realloc()*: a pointer to the start of the reallocated memory, or NULL if there isn't sufficient memory available or the *size* is set to zero.

Classification:

QNX Neutrino

Safety	
Cancellation point	No
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

ado_calloc(), *ado_free()*, *ado_malloc()*, *ado_memory_dump()*

realloc() in the QNX *Library Reference*

ado_rwlock_destroy()

Destroy a read-write lock

Synopsis:

```
#include <audio_driver.h>

ado_rwlock_destroy( pthread_rwlock_t *rwl )
```

Arguments:

rwl A pointer to the **pthread_rwlock_t** structure for the read-write lock.

Description:

The *ado_rwlock_destroy()* macro destroys the given read-write lock.

This macro is defined as *pthread_rwlock_destroy()*, or *ado_rwlock_destroy_debug()* if ADO_RWLOCK_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message to help you locate a locking problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_rwlock_destroy()*:

EOK Success.

EBUSY The read-write lock is still in use. The behavior of this read-write lock is now undefined.

Classification:

QNX Neutrino

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

See also:

ado_rwlock_init(), *ado_rwlock_rdlock()*, *ado_rwlock_unlock()*, *ado_rwlock_wrlock()*

pthread_rwlock_destroy() in the QNX *Library Reference*

ado_rwlock_init()

Initialize a read-write lock

Synopsis:

```
#include <audio_driver.h>

int ado_rwlock_init( pthread_rwlock_t *rwl);
```

Arguments:

rwl A pointer to the **pthread_rwlock_t** structure for the read-write lock.

Description:

This macro initializes the given read-write lock.

The *ado_rwlock_init()* macro is defined as *pthread_rwlock_init()*, or *ado_rwlock_init_debug()* if ADO_RWLOCK_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message to help you locate a locking problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_rwlock_init()*:

EOK	Success.
EAGAIN	Insufficient system resources to initialize the read-write lock.
EBUSY	The read-write lock *rwl* has been initialized or unsuccessfully destroyed.
EFAULT	A fault occurred when the kernel tried to access *rwl* or *attr*.
EINVAL	Invalid read-write lock attribute object *attr*.

Classification:

QNX Neutrino

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

See also:

ado_rwlock_destroy(), *ado_rwlock_rdlock()*, *ado_rwlock_unlock()*, *ado_rwlock_wrlock()*

pthread_rwlock_init() in the QNX *Library Reference*

ado_rwlock_rdlock()

Acquire a shared read lock on a read-write lock

Synopsis:

```
#include <audio_driver.h>

ado_rwlock_rdlock( pthread_rwlock_t *rwl);
```

Arguments:

rwl A pointer to the **pthread_rwlock_t** structure for the read-write lock.

Description:

This macro acquires a shared read lock on the given read-write lock.

The *ado_rwlock_rdlock macro()* is defined as *pthread_rwlock_rdlock()*, or *ado_rwlock_rdlock_debug()* if ADO_RWLOCK_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message to help you locate a locking problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_rwlock_rdlock()*:

EOK	Success.
EAGAIN	On the first use of statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.
EDEADLK	The calling thread already has an exclusive lock for *rwl*.
EFAULT	A fault occurred when the kernel tried to access *rwl*.
EINVAL	The read-write lock *rwl* is invalid.

Classification:

QNX Neutrino

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

See also:

ado_rwlock_destroy(), *ado_rwlock_init()*, *ado_rwlock_unlock()*, *ado_rwlock_wrlock()*

pthread_rwlock_rdlock() in the QNX *Library Reference*

ado_rwlock_unlock()

Unlock a read-write lock

Synopsis:

```
#include <audio_driver.h>

int ado_rwlock_unlock( pthread_rwlock_t *rwl );
```

Arguments:

rwl A pointer to the **pthread_rwlock_t** structure for the read-write lock.

Description:

This macro unlocks the given read-write lock.

The *ado_rwlock_unlock()* macro is defined as *pthread_rwlock_unlock()*, or *ado_rwlock_unlock_debug()* if ADO_RWLOCK_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message to help you locate a locking problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_rwlock_unlock()*:

EOK	Success.
EAGAIN	On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.
EFAULT	A fault occurred when the kernel tried to access *rwl*.
EINVAL	The read-write lock *rwl* is invalid.
EPERM	No thread has a read or write lock on *rwl* or the calling thread doesn't have a write lock on *rwl*.

Classification:

QNX Neutrino

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

See also:

ado_rwlock_destroy(), *ado_rwlock_init()*, *ado_rwlock_rdlock()*, *ado_rwlock_wrlock()*

pthread_rwlock_unlock() in the QNX *Library Reference*

ado_rwlock_wrlock()

Acquire an exclusive write lock on a read-write lock

Synopsis:

```
#include <audio_driver.h>

int ado_rwlock_wrlock( pthread_rwlock_t *rwl );
```

Arguments:

rwl A pointer to the **pthread_rwlock_t** structure for the read-write lock.

Description:

This macro acquires an exclusive write lock on the given read-write lock.

The *ado_rwlock_wrlock()* macro is defined as *pthread_rwlock_wrlock()*, or *ado_rwlock_wrlock_debug()*, if ADO_RWLOCK_DEBUG and ADO_DEBUG are defined. The debug version uses *ado_debug()* to display a message, to aid in locating a locking problem in the driver.

For more information, see "Debugging an audio driver" in the Organization of a Driver chapter.

Returns:

Same as *pthread_rwlock_wrlock()*:

EOK	Success.
EAGAIN	On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.
EDEADLK	The calling thread already has an exclusive lock for *rwl*.
EFAULT	A fault occurred when the kernel tried to access *rwl*.
EINVAL	The read-write lock *rwl* is invalid.

Classification:

QNX Neutrino

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

See also:

ado_rwlock_destroy(), *ado_rwlock_init()*, *ado_rwlock_rdlock()*, *ado_rwlock_unlock()*

pthread_rwlock_wrlock() in the QNX *Library Reference*

ado_shm_alloc()

Allocate shared memory

Synopsis:

```
#include <audio_driver.h>

void *ado_shm_alloc( size_t size,
                     char *name,
                     int32_t flags,
                     off64_t *phys_addr );
```

Arguments:

size	The size of the block to allocate, in bytes.
name	The name of the shared region. The storage requirements for *name* are defined as QNX_SHM_NAME_LEN.
flags	Restrictions on the memory region; any combination of: • ADO_SHM_DMA_SAFE — the region must be physical to allow DMA operations. • ADO_DMA_ISA — the region must begin on a 64-kilobyte boundary. • ADO_DMA_16M — the region must be allocated within the first 16M of memory.
phys_addr	A pointer to where *ado_shm_alloc()* should store the physical address of the region.

Description:

The *ado_shm_alloc()* function allocates a block of shared memory. You use it mostly for the DMA buffer so that it can be shared.

The *ado_shm_mmap()* function is for cards where the DMA buffer is in onboard memory; it mmaps the card memory and makes the region shared without allocating it.

Returns:

A pointer to the allocated memory, or NULL if an error occurred (*errno* is set).

Classification:

QNX Neutrino

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

See also:

ado_shm_free(), *ado_shm_mmap()*

ado_shm_free()

Release memory and unlink shared memory

Synopsis:

```
#include <audio_driver.h>

void ado_shm_free( void *ptr,
                   size_t size,
                   char *name );
```

Arguments:

ptr	A pointer to the shared memory.
size	The size of the shared memory.
name	The name of the shared memory region.

Description:

The *ado_shm_free()* function releases the memory at the location *ptr* and of size *size*. Additionally, it unlinks the shared memory region of *name*.

Classification:

QNX Neutrino

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

See also:

ado_shm_alloc()

ado_shm_mmap()

Map a shared memory region into the Audio HW DLL's address space

Synopsis:

```
#include <audio_driver.h>

void *ado_shm_mmap( off64_t phys_addr,
                    size_t size,
                    char *name );
```

Arguments:

phys_addr	The physical address of the region.
size	The size of the region.
name	The name of the region. The storage requirements for *name* are defined as QNX_SHM_NAME_LEN.

Description:

The *ado_shm_mmap()* function maps a shared memory region into the Audio HW DLL's address space. This function is for cards where the DMA buffer is in onboard memory; it mmaps the card memory and makes the region shared without allocating it.

The *ado_shm_alloc()* function allocates a shared memory region; you use it mostly for the DMA buffer so that it can be shared.

Returns:

A pointer to the shared memory region, or NULL if an error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	No

See also:

ado_shm_alloc(), ado_shm_free()

ado_strdup()

Create a duplicate of a string

Synopsis:

```
#include <audio_driver.h>

char *ado_strdup_debug( const char *src );
```

Arguments:

src The string to be copied.

Description:

The *ado_strdup()* macro creates a copy of the string, *src*.

This macro is defined as *strdup()*, or *ado_strdup_debug()* if ADO_DEBUG is defined; see "Debugging an audio driver" in the Organization of a Driver chapter.

The advantage of using the debug variant is that it tracks the memory allocated until it's freed; see *ado_memory_dump()*.

Returns:

Same as *strdup()*: a pointer to a copy of the string for success, or NULL.

Classification:

QNX Neutrino

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

See also:

ado_calloc(), *ado_free()*, *ado_malloc()*, *ado_memory_dump()*, *ado_realloc()*

strdup() in the QNX *Library Reference*

dma_interrupt()

Signal that the current fragment of a subchannel has been completed by the DMA engine

Synopsis:

```
#include <audio_driver.h>

void dma_interrupt( ado_pcm_subchn_t *subchn );
```

Arguments:

subchn A pointer to the **ado_pcm_subchn_t** structure that describes the subchannel.

Description:

The *dma_interrupt()* function signals to **io-audio** that the current fragment of subchannel *subchn* has been completed by the DMA engine. The upper layer uses this information for time synchronization and to determine if the stream should continue or stop.

Classification:

QNX Neutrino

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

Appendix A

Supported Codecs

In this appendix. . .

This appendix contains the list of supported codecs for the QNX Audio Architecture. A supported codec in these terms is a codec for which QNX Software Systems (QSS) has produced a Dynamic Linked Library (DLL) to control it.

The currently supported codecs include:

- Audio Codec 97 (AC97)
- AK4531

For QSS to support a codec, it must be a standardized part with a detailed specification, and must be used in audio platforms of use to QSS and its customers.

As well as providing analog audio mixer functions, a codec may also provide other callback functions of use to the audio driver, such as Sample Rate Conversion (SRC) or S/PDIF. Handling PCM Audio Data chapter.

To use one of these codecs, call *ado_mixer_dll()* in your Audio HW DLL's *ctrl_init()* function. For more information, see "Using a standard mixer DLL" in the Handling Analog Audio Data chapter.

The sections below indicate what arguments you need to pass to *ado_mixer_dll()*.

Audio Codec 97 (AC97)

DLL Name

Pass **"ac97"** as the *mixer_dll* argument to *ado_mixer_dll()*.

Header File

<mixer/ac97_dll.h>

Parameter Structure

This structure holds pointers to the callback functions that you need to provide for the mixer DLL to call:

```
typedef
struct  ado_mixer_dll_params_ac97
{
  HW_CONTEXT_T  *hw_context;
  uint16_t      (*read) (HW_CONTEXT_T *hw_context,
                         uint16_t reg);
  void          (*write) (HW_CONTEXT_T *hw_context,
                          uint16_t reg, uint16_t val);
  int32_t       (*init) (HW_CONTEXT_T *hw_context);
  void          (*destroy) (HW_CONTEXT_T *hw_context);
} ado_mixer_dll_params_ac97_t;
```

The members include:

hw_context	A pointer to a structure that you can use for any context-sensitive data that your driver needs. It's used only as an argument for the function calls.
read()	Read the given ac97 register.
write()	Write the given value into the given ac97 register.
init()	The upper layer of the driver calls this function after the codec has been initialized. Use this callback to make any last-minute adjustments.
destroy()	The upper layer of the driver calls this function just before the DLL terminates. In general, you should use it to undo anything *init()* did.

Pass a pointer to a **`ado_mixer_dll_params_ac97_t`** structure as the *params* argument to *ado_mixer_dll()*.

Supported Device Controls

This structure holds pointers to the callback functions, provided by the mixer DLL, that your Audio HW DLL can call to control the device:

```
enum ac97_SRC
{
  AC97_FRONT_DAC_SRC,
  AC97_SURR_DAC_SRC,
```

```
  AC97_LFE_DAC_SRC,
  AC97_LR_ADC_SRC,
};

typedef
struct  ado_mixer_dll_callbacks_ac97
{
  MIXER_CONTEXT_T  *mix_context;
  int32_t          (*SRC_test) (MIXER_CONTEXT_T * mix_context,
                                enum ac97_SRC src,
                                ado_pcm_cap_t * caps);
  int32_t          (*SRC_set) (MIXER_CONTEXT_T * mix_context,
                               enum ac97_SRC src,
                               uint32_t * rate);
}
ado_mixer_dll_callbacks_ac97_t;
```

The members include:

mix_context — A pointer to a structure that the codec uses for any context-sensitive data that it needs. Use it only as an argument for the function calls defined in this structure.

SRC_test — Checks which sample rates are supported for the *src* sample rate converter and updates the information in the given **ado_pcm_cap_t** structure.

SRC_set — Sets the sample rate for the *src* sample rate converter.

Pass a pointer to a **ado_mixer_dll_callbacks_ac97_t** structure as the *callbacks* argument to *ado_mixer_dll()*.

References

http://developer.intel.com/ial/scalableplatforms/audio/index.htm

ftp://download.intel.com/ial/scalableplatforms/ac97r22.pdf

AK4531

DLL Name

Pass **"ak4531"** as the *mixer_dll* argument to *ado_mixer_dll()* function call.

Header File

<mixer/ak4531_dll.h>

Parameter Structure

This structure holds pointers to the callback functions that you need to provide for the mixer DLL to call:

```
typedef
struct  ado_mixer_dll_params_ak4531
{
  HW_CONTEXT_T  *hw_context;
  void          (*write) (HW_CONTEXT_T *hw_context,
                          uint16_t reg, uint16_t val);
  void          (*destroy) (HW_CONTEXT_T *hw_context);
} ado_mixer_dll_params_ak4531_t;
```

The members include:

hw_context	A pointer to a structure that you can use for any context-sensitive data that your driver needs. It's used only as an argument for the function calls.
write()	Write the given value into the given ak4531 register.
destroy()	The upper layer of the driver calls this function just before the DLL terminates.

Pass a pointer to a **ado_mixer_dll_params_ak4531_t** structure as the *params* argument to *ado_mixer_dll()*.

Supported Device Controls

None; pass a NULL pointer as the *callbacks* argument to *ado_mixer_dll()*.

Reference

http://www.akm.com/ProductPages/ak4531.html

Appendix B

Sample Mixer Source

```
/*
 *  example_mixer.c
 *  The primary interface into the sample mixer.
 *  24 Jan 2001
 *  Copyright QNX Software Systems Ltd, all rights reserved
 */

#include <example.h>
#include <proto.h>

static  uint8_t
example_mixer_read (example_t * example, uint32_t reg)
{
 uint8_t val = 0;

 /* Generic code to read a mixer register and set
  * the variable "val" should go here.
  */

 return val;
}

static void
example_mixer_write (example_t * example, uint32_t reg, uint8_t val)
{
 /* Generic code to write a mixer register with the value of
  * the variable "val" should go here.
  */
}

static snd_mixer_voice_t stereo_voices[2] =
{
 {SND_MIXER_VOICE_LEFT, 0},
 {SND_MIXER_VOICE_RIGHT, 0}
};

static struct snd_mixer_element_volume1_range output_range[2] =
{
 {0, 63, -600, 1650},
 {0, 63, -600, 1650}
};

static struct snd_mixer_element_volume1_range input_range[2] =
{
 {0, 15, -2850, 300},
 {0, 15, -2850, 300}
};

static  int32_t
example_master_vol_control (MIXER_CONTEXT_T * example,
```

```
  ado_mixer_delement_t * element, uint8_t set, uint32_t * vol,
  void *instance_data)
{
 enum example_mixer_reg reg = (int32_t) instance_data;
 uint32_t data[2];
 int32_t altered = 0;
 int     max = ado_mixer_element_vol_range_max (element);

 data[0] = example_mixer_read (example, reg + 0);
 data[1] = example_mixer_read (example, reg + 2);
 if (set)
 {
  altered = vol[0] != ( data[0] & max ) || vol[1] != ( data[1] & max );
  data[0] = ( data[0] & ~(max) ) | vol[0];
  data[1] = ( data[1] & ~(max) ) | vol[0];
  example_mixer_write (example, reg + 0, data[0]);
  example_mixer_write (example, reg + 2, data[1]);
 }
 else
 {
  vol[0] = (data[0] & max);
  vol[1] = (data[1] & max);
 }

 return (altered);
}

static  int32_t
example_master_mute_control (MIXER_CONTEXT_T * example,
  ado_mixer_delement_t * element, uint8_t set, uint32_t * val,
  void *instance_data)
{
 enum example_mixer_reg reg = (int32_t) instance_data;
 uint32_t data[2];
 int32_t altered = 0;

 data[0] = example_mixer_read (example, reg + 0);
 data[1] = example_mixer_read (example, reg + 2);
 if (set)
 {
  altered = val[0] != ( ( data[0] & 0x40 ) ? ( 1 << 0) : 0 ) |
                      ( ( data[1] & 0x40 ) ? ( 1 << 1) : 0 );
  data[0] = ( data[0] & ~0x40 ) | ( ( val[0] & (1 << 0) ) ? 0x40 : 0 );
  data[1] = ( data[1] & ~0x40 ) | ( ( val[0] & (1 << 1) ) ? 0x40 : 0 );
  example_mixer_write (example, reg + 0, data[0]);
  example_mixer_write (example, reg + 2, data[1]);
 }
 else
  val[0] = ( ( data[0] & 0x40 ) ? (1 << 0) : 0 ) |
           ( ( data[1] & 0x40 ) ? (1 << 1) : 0 );
```

```
 return (altered);
}

static  int32_t
example_vol_control (MIXER_CONTEXT_T * example,
  ado_mixer_delement_t * element, uint8_t set, uint32_t * vol,
  void *instance_data)
{
 enum example_mixer_reg reg = (int32_t) instance_data;
 uint32_t data;
 int32_t altered = 0;
 int     max = ado_mixer_element_vol_range_max (element);

 data = example_mixer_read (example, reg);
 if (set)
 {
  altered = vol[0] != ( ( data & (max << 8) ) >> 8 ) || vol[1] != (data & max);
  data = (data & ~( (max << 4) | max) ) | ( (vol[0]) <<  4) | (vol[1]);
  example_mixer_write (example, reg, data);
 }
 else
 {
  vol[0] = ( (data & (max << 4) ) >> 4);
  vol[1] = ( (data & max) );
 }

 return (altered);
}

int
build_in_group( MIXER_CONTEXT_T * example, char *name_p, char *name_c,
  example_group * grp, enum example_mixer_reg reg_p,
  enum example_mixer_reg reg_c)
{
 int     error = 0;
 ado_mixer_delement_t *pre_elem, *elem = NULL;
 char    ename[sizeof (((snd_mixer_eid_t *) 0)->name)];


 if ( grp == &example->pcm )
 {
  if ( !error && (elem = ado_mixer_element_io (example->mixer, name_p,
       SND_MIXER_ETYPE_PLAYBACK1, 0, 2, stereo_voices)) == NULL )
   error++;
 }
 else
 {
  if ( !error && (elem = ado_mixer_element_io (example->mixer, name_p,
       SND_MIXER_ETYPE_INPUT, 0, 2, stereo_voices)) == NULL )
```

```
  error++;
 }
 pre_elem = elem;

 sprintf (ename, "%s %s", name_p, "Volume");

 if ( !error && (elem = ado_mixer_element_volume1 (example->mixer, ename,
      2, input_range, example_vol_control, (void *) reg_p, NULL)) == NULL )
  error++;

 if ( !error && ado_mixer_element_route_add (example->mixer, pre_elem, elem) != 0 )
  error++;

 grp->vol_out = elem;

 if ( !error && ado_mixer_element_route_add (example->mixer, elem,
      example->output_accu) != 0 )
  error++;

 if (name_c)
 {
  if ( !error && ado_mixer_element_route_add (example->mixer, pre_elem,
       example->input_accu) != 0 )
   error++;
 }

 if ( !error && name_p && (grp->group_p = ado_mixer_playback_group_create (example->mixer,
      name_p, SND_MIXER_CHN_MASK_STEREO, grp->vol_out, NULL)) == NULL )
  error++;

 if ( !error && name_c && (grp->group_c = ado_mixer_capture_group_create (example->mixer,
      name_c, SND_MIXER_CHN_MASK_STEREO, NULL, NULL, NULL, NULL)) == NULL )
  error++;

 return (0);
}

int
build_example_mixer (MIXER_CONTEXT_T * example, ado_mixer_t * mixer)
{
 int     error = 0;
 ado_mixer_delement_t *pre_elem, *elem = NULL;


 /* ################ */
 /* the OUTPUT GROUP */
 /* ################ */
 if ((example->output_accu = ado_mixer_element_accu1 (mixer,
    SND_MIXER_ELEMENT_OUTPUT_ACCU, 0)) == NULL)
  error++;
```

```
pre_elem = example->output_accu;

if ( !error && (elem = ado_mixer_element_volume1 (mixer, "Output Volume",
     2, output_range, example_master_vol_control,
    (void *) EXAMPLE_MASTER_LEFT, NULL)) == NULL )
 error++;

if ( !error && ado_mixer_element_route_add (mixer, pre_elem, elem) != 0 )
 error++;

example->master_vol = elem;
pre_elem = elem;

if ( !error && (elem = ado_mixer_element_sw2 (mixer, "Output Mute",
     example_master_mute_control,
     (void *) EXAMPLE_MASTER_LEFT, NULL)) == NULL )
 error++;

if ( !error && ado_mixer_element_route_add (mixer, pre_elem, elem) != 0 )
 error++;

example->master_mute = elem;
pre_elem = elem;

if ( !error && (elem = ado_mixer_element_io (mixer, "Output",
     SND_MIXER_ETYPE_OUTPUT, 0, 2, stereo_voices)) == NULL )
 error++;

if ( !error && ado_mixer_element_route_add (mixer, pre_elem, elem) != 0 )
 error++;

if ( !error && (example->master_grp = ado_mixer_playback_group_create (mixer,
     SND_MIXER_MASTER_OUT, SND_MIXER_CHN_MASK_STEREO, example->master_vol,
     example->master_mute)) == NULL )
 error++;


/* ################ */
/* the INPUT GROUP */
/* ################ */
if ( (example->input_accu = ado_mixer_element_accu1 (mixer,
     SND_MIXER_ELEMENT_INPUT_ACCU, 0)) == NULL )
 error++;
pre_elem = example->input_accu;

if ( !error && (elem = ado_mixer_element_volume1 (mixer, "Input Volume", 2,
     input_range, example_vol_control,
     (void *) EXAMPLE_RECORD_LEVEL, NULL)) == NULL )
 error++;
```

```
 if ( !error && ado_mixer_element_route_add (mixer, pre_elem, elem) != 0 )
  error++;

 example->master_vol = elem;
 pre_elem = elem;

 if ( !error && (elem = ado_mixer_element_io (mixer, SND_MIXER_ELEMENT_CAPTURE,
      SND_MIXER_ETYPE_CAPTURE1, 0, 2, stereo_voices)) == NULL )
  error++;

 if ( !error && ado_mixer_element_route_add (mixer, pre_elem, elem) != 0 )
  error++;

 if ( !error && (example->input_grp = ado_mixer_capture_group_create (mixer,
      SND_MIXER_GRP_IGAIN, SND_MIXER_CHN_MASK_STEREO,
      example->master_vol, NULL, NULL, NULL)) == NULL )
  error++;


 /* ################ */
 /* the INPUT GROUPS */
 /* ################ */
 if ( !error && build_in_group (example, SND_MIXER_PCM_OUT, NULL, &example->pcm,
      EXAMPLE_PCM_OUT_VOL, NULL) != 0 )
  error++;

 if ( !error && build_in_group (example, SND_MIXER_MIC_OUT, SND_MIXER_MIC_IN,
      &example->mic, EXAMPLE_MIC_OUT_VOL, EXAMPLE_MIC_IN_VOL) != 0 )
  error++;

 if ( !error && build_in_group (example, SND_MIXER_CD_OUT, SND_MIXER_CD_IN,
      &example->cd, EXAMPLE_CD_OUT_VOL, EXAMPLE_CD_IN_VOL) != 0 )
  error++;

 return (0);
}

ado_mixer_reset_t example_reset;
int
example_reset (MIXER_CONTEXT_T * example)
{
 /* This function, if included, should restore the mixer to a default state */

 example_mixer_write( example, EXAMPLE_PCM_OUT_VOL, 0xff );   /* set PCM vol 100% */
 example_mixer_write( example, EXAMPLE_CD_OUT_VOL, 0xff );   /* set cd vol 100% */
 example_mixer_write( example, EXAMPLE_REC_SEL, 0x05 );    /* set record src to mixer */
 return (0);
}
```

```
ado_mixer_destroy_t example_destroy;
int
example_destroy (MIXER_CONTEXT_T * example)
{
 /* This function, if included, should set the mixer to a safe state */

 example_mixer_write( example, EXAMPLE_PCM_OUT_VOL, 0x0 );    /* set PCM vol 0% */
 example_mixer_write( example, EXAMPLE_CD_OUT_VOL, 0x00 );    /* set cd vol 0% */
 return (0);
}

int
example_mixer (ado_card_t * card, HW_CONTEXT_T * example)
{
 int32_t status;

 if ( (status = ado_mixer_create (card, "Example", &example->mixer, example)) != EOK )
  return (status);

 example_mixer_write (example, 0x00, 0x00);       /* reset the mixer */

 if ( build_example_mixer (example, example->mixer) )
  return (-1);

 if ( example_reset (example) )
  return (-1);

 /* The following functions are optional, but if you have actions
  * that should be performed by the hardware whenever the mixer is
  * reset or destroyed. These functions are specifically for
  * hardware specific requirements.
  */

 ado_mixer_set_reset_func( example->mixer, example_reset );

 ado_mixer_set_destroy_func( example->mixer, example_destroy );

 return (0);
}
```

Glossary

ADC

Analog Digital Converter. This converts analog audio signals into a digital stream of samples.

ALSA

Advanced Linux Sound Architecture.

capture group

A mixer group that contains up to one volume, one mute, and one input selection element.

CD

Compact disk.

codec

Compression-Decompression module.

DAC

Digital Analog Converter. This converts a digital stream of samples into analog signals.

MIC

Microphone.

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.

S/PDIF

Sony / Philips Digital InterFace. A hardware and protocol standard for the transmission of digital audio signals.

SRC

Sample Rate Conversion.

Index

A

C

D

E

H

I

L

M

P

R

S

T