Using the Sound Driver Interface

Version 3.2
Copyright and publication information

This manual reflects version 3.2 of MAUI.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.
Chapter 1: Sound Concepts

8 Overview
9 Entry Points
9 Sound Functions and Data Types
10 Sound Maps
12 Triggers and Status
13 Encoding Parameters
14 Sound Buffer
15 Looping
15 Next Sound Map
16 Preparing to Use Sound
16 Device Capabilities
16 Example 1
17 Example 2
17 SND_DEV_CAP Data Structure
20 Driver Compatibility Level

Chapter 2: Sound Operations

22 Preparing the Sound Device for Use
22 Get the Sound Device Name
22 Open
23 Get Device Capabilities
24 Keeping Track of the Sound Device
24 Sound Device Status
25 Status
25 Gain
25 Send Signal
Chapter 3: Function Reference

Chapter 4: Data Type Reference

Using the Sound Driver Interface
80 SND_CM_PCM_SLINEAR, SND_CM_PCM_SLINEAR | SND_CM_LSBYTE1ST  
SND_CM_PCM_ULINEAR  
SND_CM_PCM_ULINEAR | SND_CM_LSBYTE1ST  
84 SND_CM_PCM_ULAW  
SND_CM_PCM_ALAW  
85 SND_CM_ADPCM_IMA  
86 SND_CM_ADPCM_G721  
87 SND_CM_ADPCM_G723  
87 SND_CM_OEM_*

Appendix A: Sound Hardware Specifications

120 Crystal Semiconductor CS4231A
120 Overview
121 Device Capabilities
123 Gain Capabilities Array
134 Sample Rates
134 Number of Channels
135 Encoding and Decoding Formats

Index
Chapter 1: Sound Concepts

The Sound Driver Interface enables your MAUI® application to play and record sound data. This chapter explains the concepts of using sound in your applications.
Overview

The Sound Driver Interface:

- provides a set of primary entry points, GetStat sub-functions, and SetStat sub-functions through which MAUI applications can control the sound driver.
- is a dual-ported I/O (DPIO) driver that uses the multimedia file manager (MFM). This allows the driver to work under both OS-9® and OS-9 for 68K.

The sound driver is sharable (multiple paths may be open to it at the same time). This enables multiple play and record paths, but not concurrent play and record.

The Sound Driver Interface is accessible by MAUI applications and directly controls the operation of the sound driver as shown in the following figure.

Figure 1-1  Sound Driver Interface Architecture
Entry Points

The primary entry points into the sound driver are those found in all DPIO drivers. They are `init`, `terminate`, `open`, `close`, `getstat`, and `setstat`. The `getstat` and `setstat` entry points are dispatch functions that call the proper sub-function based on the information in the parameter block passed to it.

Sound Functions and Data Types

The Sound Driver Interface provides the functions your application needs to play and record sound. The majority of functions can be loosely divided into two categories: get status functions and set status functions.

Get status functions are identified by the `_os_gs_*` name construction. These functions are used by the application to get information or status from the sound driver. Your application uses get status functions to determine sound device status and capabilities.

Set status functions are identified by the `_os_ss_*` name construction. They are used by the application to send control commands or set the status of the sound driver. Your application uses set status functions to instruct the sound driver to send signals when the device is ready, play sound data, record sound, and release the device when it is no longer needed.

Sound data types comprise constants, enumerated types, and data structures that provide information to the application and the sound driver. Data types control how sound drivers operate and are fundamental to hardware independence.
Sound Concepts

Sound Maps

Sound map objects are created by the application to control play and record operations. Sound maps contain all the information needed by the sound driver to play and record sound data. Your application may define one or more sound maps depending on the type of operations and type of sound data used.

Play and record operations can be synchronous or asynchronous. The driver generates signals for the application to keep the application informed about the operation. Since the sound device can be shared by multiple processes, these signals can also enable your application to manage multiple play and record requests.

Sound maps used for play operations are built by the application at run time with the information contained in the headers of the sound data files. Sound maps used for recording operations are initialized and defined by the application through standard C syntax.

Sound data is contained in buffers and played from buffers. Sound map looping enables your application to loop segments of sound data during play, or play a segment of sound data rather than an entire sequence from start to finish.

The sound map can also contain a pointer to another sound map. This pointer enables you to link sound maps together for continuous operation from a single play or record command or link sound maps of different formats.
The sound map is defined by the data structure `SND_SMAP` and contains the following information:

- **Triggers and status**: Trigger signals and status information are sent to the application to indicate the current sound device conditions.
- **Encoding parameters**: Sound data encoding method, number of channels, bits per sample, and sample rate of the sound data.
- **Sound Buffer**: Pointer or address of the buffer, buffer size, and current offset.
- **Looping**: Loop start offset, end offset, loop count, and current loop counter value.
- **Next sound map**: Pointer to the next sound map to play or record.

Each play and record operation is associated with one or more sound maps. Your application may have a single sound map defined for play and a single sound map defined for record, or you may have a different sound map defined for each chunk of sound data your application may play or record. Since each sound map is associated with a single buffer, the size of the buffer determines how much sound data can be stored at any time. If your application uses sound data that is of the same general size, a single buffer and sound map may be appropriate. If your application uses sound data of various sizes, or if your application actively manipulates the sound data by using offsets and looping, you may want to define a larger number of small buffers.

Sound maps may be linked together to form a chain of sound maps to play or record. These sound maps do not have to contain data of the same encoding formats and parameters. When sound maps of different encoding formats and parameters are linked, you may notice some delay when transitioning from one encoding format to another as the driver calibrates to the new parameters. This depends on the characteristics of the hardware.
Triggers and Status

The sound map data structure defines the triggers and status used during the play and recording operations.

Trigger status  
The variable \texttt{trig\_status} indicates the sound map status. The trigger status is modified as the status changes. The data structure \texttt{SND\_TRIGGER} describes each of the possible status values.

Trigger mask  
The variable \texttt{trig\_mask} indicates which activities prompt the sending of a trigger signal. If the trigger mask is set to \texttt{SND\_TRIG\_NONE}, no signals are sent.

Trigger signal  
The variable \texttt{trig\_signal} defines the signal to send when the sound operation completes an activity specified by the trigger mask. If the trigger signal is set to zero, no signals are sent.

Error code  
The variable \texttt{err\_code} contains the exit status for the sound map. If an error occurs, such as \texttt{EOS\_MAUI\_NOHWSUPPORT} or \texttt{EOS\_MAUI\_ABORT}, the error code is placed in \texttt{errno}. Otherwise, the error code is set to zero at the end of the sound operation.
**Encoding Parameters**

The encoding parameters in `SND_SMAP` define the specifics of the sound data.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Coding method</strong></td>
<td>The variable <code>coding_method</code> defines the audio encoding method and format for the sound data attached to the sound map.</td>
</tr>
<tr>
<td><strong>Number of Channels</strong></td>
<td>The variable <code>num_channels</code> specifies the number of audio channels. Mono data requires one channel, stereo data requires two channels.</td>
</tr>
<tr>
<td><strong>Sample size</strong></td>
<td>The variable <code>sample_size</code> indicates the number of bits per sample. Typically, this value is 8 or 16 bits per sample.</td>
</tr>
<tr>
<td><strong>Sample rate</strong></td>
<td>The variable <code>sample_rate</code> indicates the number of samples per second. Typical sample rates include 4000, 8000, 11025, 22050, 44100, and 48000 samples per second.</td>
</tr>
</tbody>
</table>

Different sound devices support a variety of encoding parameters. Sound maps contain the information required by sound devices to properly calibrate to the parameters of various encoding parameters.

The two parameters that most affect sound quality are the sample size and the sample rate. Higher sample rates provide a higher density of information in the sound data. Higher sample sizes provide greater dynamic range and a more favorable signal to noise ratio. Alternative coding methods such as ADPCM provide a means to represent the same audio data in smaller sample sizes.
### Sound Buffer

The actual sound data is stored in a simple buffer.

**Buffer Pointer**

The entry `*buf` points to the buffer containing the sound data. The starting point of the buffer should be on a multiple of `cm_boundary_size`. `cm_boundary_size` is defined in the structure `SND_DEV_CM`.

**Buffer size**

The variable `buf_size` indicates the size of the sound data buffer in bytes. The size of the buffer should be a multiple of `cm_boundary_size`.

**Current offset**

The variable `cur_offset` specifies the offset in bytes within the buffer for the current sound operation. The offset is updated throughout the I/O operation by the driver. The application should set the offset to the play or record starting point. This should be set to a value consistent with `cm_boundary_size`. 
Looping

Looping entries are used for play operations to specify the area of the buffer to play and how many times to play the buffer area. These entries must be set to zero for recording operations.

Loop start
The variable *loop_start* specifies the start of the loop expressed as an offset in bytes from the start of the buffer. This value should be consistent with *cm_boundary_size*.

Loop end
The variable *loop_end* specifies the end of the loop expressed as an offset in bytes from the start of the buffer. This value should be consistent with *cm_boundary_size*.

Loop count
The variable *loop_count* specifies the number of times to execute the loop.

Loop counter
The variable *loop_counter* indicates the current number of times the loop has executed. This entry is set to zero when the sound map is accepted by the sound driver and incremented as the driver plays the sound data.

Next Sound Map

The final entry in the *SND_SMAP* data structure, *next*, is a pointer to another sound map. *next* identifies the next sound map that is used when the current sound map is finished. This entry is *NULL* if this sound map is not linked to another sound map.

Linked sound maps may have different encoding parameters from each other. Since most hardware requires some calibration time to switch to a different format, there may be some delay before a sound map of a different format begins to play or record.
Preparation to Use Sound

MAUI applications are likely to run on a variety of user devices, so the application must evaluate its environment and adjust its operating parameters appropriately before actually playing and recording sound. Three functions enable your application to evaluate the software and hardware environment. They are:

- Check device capabilities `os_gs_snd_devcap()` determines the capabilities of the sound hardware and driver.
- Check software compatibility `os_gs_snd_compat()` determines that the compatibility level of the sound driver and sound driver interface are functionally compatible.
- Check the status `os_gs_snd_status()` determines the current status of the sound device.

**Device Capabilities**

The specific set of sound device capabilities is stored in a data structure called `SND_DEV_CAP`. The getstat sub-function `os_gs_snd_devcap()` reads the data structure `SND_DEV_CAP` and returns the information to the application. Based on the information returned, your applications can make adjustments to run correctly on the target system. Following are two examples of how an application can deal with differing hardware capabilities at run time.

**Example 1**

When you design your application you must be flexible since you may not know which specific sound board and processor are installed in the user hardware. Consequently, you cannot know which audio file formats are supported. To deal with this unknown, you can encode your sound data in several file formats. Give each file the same base name, but store each set of files in a different directory, such as `PCM8/intro.snd` and `ADPCM/intro.snd`. 

...
When your application plays sound data, it uses a variable as part of the path name (%s/intro.snd). At run time, in this example, your application reads the SND_DEV_CAP and discovers that the user hardware supports 8-bit PCM audio. At that time, your application adjusts to the hardware by setting the %s variable to PCM8. When your application plays the sound data, it gets the data from the PCM8 directory.

You can use this same technique to adjust to mono and stereo options and sample rates.

**Example 2**

An application is designed to play a sound sample that was recorded at 22000 Hz, but that specific sampling rate is not supported on the playback device. The application can examine the sample rates section of the SND_DEV_CAP to determine the supported sampling rate that is closest to 22000 Hz and convert the original sound data to match the nearest supported sample rate or play the sample back at a supported rate.

Applications can provide built-in conversion utilities for mono and stereo options, gain control, sample rates, or coding method.

**SND_DEV_CAP Data Structure**

SND_DEV_CAP contains the following device capabilities information:

- Hardware type
- Hardware subtype
- Mask of supported triggers
- Gain controls for play operations
- Gain controls for record operations
- Mask of supported gain commands
- Number and pointer to a list of gain capabilities
18 Using the Sound Driver Interface

Sound Concepts

- Number and pointer to a list of supported sample rates
- Number of channel entries and pointer to a list of supported channels
- Number and pointer to a list of supported coding methods

The hardware type and subtype describe the specific sound hardware installed in the user system. Hardware type indicates the general class of the hardware, such as the type of sound chip. Hardware subtype typically indicates the specific sound board installed in the system.

The sound triggers entry defines the trigger events in a sound operation. The following table describes the six trigger events and how they are generated.

Table 1-1 Trigger Signals

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SND_TRIG_NONE</td>
<td>Mask value indicates to send no trigger signals to the application.</td>
</tr>
<tr>
<td>SND_TRIG_ANY</td>
<td>Mask value indicates to send all trigger signals to the application.</td>
</tr>
<tr>
<td>SND_TRIG_START</td>
<td>Trigger value indicates to send a trigger signal to the application when the output device is actively playing or has started accepting sound input.</td>
</tr>
<tr>
<td>SND_TRIG_FINISH</td>
<td>Trigger value indicates to send a trigger signal to the application when the output device has completed playing or stopped accepting sound input.</td>
</tr>
</tbody>
</table>
Digital sound data can be classified by coding method and sample rate. The list that follows presents some commonly supported coding methods for sound data. For more specific information about these coding methods, see the data type reference section.

<table>
<thead>
<tr>
<th>Coding Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>μ-Law</td>
<td>Pronounced mu-law. 8-bit companded format that is a telephony standard for the United States and Japan. (Pronounced mu-law)</td>
</tr>
<tr>
<td>A-Law</td>
<td>8-bit companded format that is a telephony standard for Europe.</td>
</tr>
<tr>
<td>PCM Linear</td>
<td>Pulse-Coded Modulation. 8-bit and 16-bit format that is the most common method used by sound cards for record and playback.</td>
</tr>
<tr>
<td>ADPCM</td>
<td>Adaptive Differential Pulse Code Modulation is a compressed form of PCM coding. There are several different ADPCM standards, each with their own type code.</td>
</tr>
</tbody>
</table>
Each coding method supports one or more sample rates. Some of the sample rates may employ different formatting options. For example, 16-bit signed linear PCM is formatted either MSB first (SND_CM_MSBYTE1ST, big-endian) or LSB first (SND_CM_LSBYTE1ST, little-endian). Some sound devices accept either format, while other sound devices support only one format.

**Driver Compatibility Level**

Typically, the compatibility interface is used by the MAUI APIs to compare the compatibility level of the driver to the managing API. The newer entity of the two determines if it can operate with the other entity. If they are compatible, the application operates without problem. If they are not compatible, the newer entity either copes with the differences, or simply rejects the operation by returning an error. Since the sound driver does not currently have a high-level MAUI Sound API, this call may be used by an application to cope with compatibility differences between the Sound Driver Interface and the driver. MAUI applications may determine their compatibility level by examining the value of MAUI_COMPAT_LEVEL, which is available when you include maui_com.h.
Chapter 2: Sound Operations

This chapter provides information about the functions that are available to your applications and how to play and record sound.
Preparing the Sound Device for Use

Before sending to or receiving sound data from the sound device, your application must get the name of the sound device, open the device for use, and then make needed adjustments to the specific capabilities of the sound device.

- **cdb_get_ddr()**: Returns the name of the sound device.
- **_os_open()**: Opens a path to the sound device.
- **_os_gs_snd_devcap()**: Returns the device capabilities of the sound device.

Get the Sound Device Name

Use the CDB function `cdb_get_ddr()` with `CDB_TYPE_SOUND` to get the name of the sound device.

For More Information

`cdb_get_ddr()` and `CDB_TYPE_SOUND` are explained in detail in the MAUI Programming Reference Manual.

Open

The function `_os_open()` establishes a path ID to the named sound device for the process that opened the device. All subsequent sound control commands issued by that process use the path ID to address the sound device.

The `mode` parameter in the `_os_open()` command indicates the operational mode of the device for this process. For play operations, the device must be opened for write (mode `S_IWRITE`). For record operations, the device must be opened for read (mode `S_IREAD`).
Get Device Capabilities

The function `_os_gs_snd_devcap()` returns the sound device capabilities to the application. The device capabilities enable the application to know which sound operations are supported and what sound data encoding parameters are valid. With this information the application can make adjustments to the hardware and software environment of the target system.
Keeping Track of the Sound Device

Before calling any sound operations, your application should set up the signals and triggers that it needs to manage sound operations effectively. There are three functions your application can use to monitor and control the sound device operational status:

- `os_gs_snd_status()` - Gets the sound device status.
- `os_ss_sendsig()` - Sets up a signal for the driver to send to a process when the driver is ready for a new operation.
- `os_ss_relea()` - Removes the signal set by `os_ss_sendsig()`.

Sound Device Status

The status of the sound device can be checked at any time by using the function `os_gs_snd_status()`. This function returns a pointer to the `SND_DEV_STATUS` structure. The `SND_DEV_STATUS` structure provides information about what the sound device is doing and which process or processes are currently using the sound device.

<table>
<thead>
<tr>
<th>Status</th>
<th>Indicates the general status of the sound device.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play PID</td>
<td>Process ID of the current Play operation.</td>
</tr>
<tr>
<td>Record PID</td>
<td>Process ID of the current Record operation.</td>
</tr>
<tr>
<td>Gain</td>
<td>Pointer to an array of <code>num_gain SND_GAIN</code> structures. These describe the current settings of each gain control.</td>
</tr>
</tbody>
</table>
### Status

The status member of `SND_DEV_STATUS` is of type `SND_STATUS`. Six status types are defined in the enumerated type `SND_STATUS`.

- **SND_STATUS_NONE**
  This status indicates the sound device is in the initial state.

- **SND_STATUS_PLAY**
  This status indicates a play operation is active.

- **SND_STATUS_PLAY_PAUSED**
  This status indicates the current play operation is paused.

- **SND_STATUS_RECORD**
  This status indicates a record operation is active.

- **SND_STATUS_RECORD_PAUSED**
  This status indicates a record operation is paused.

- **SND_STATUS_BUSY**
  This status indicates the driver is processing a sound operation.

### Gain

The gain member of `SND_DEV_STATUS` points to an array of `SND_GAIN` structures that contain the current settings of the device’s various mixer lines.

### Send Signal

The function `_os_ss_sendsig()` sets up a signal to send to a process when the sound device is ready to accept a new sound operation (play or record). This function is especially useful when several processes share the sound device, so a process can be signalled when the sound device becomes idle.

Each time the signal is sent, this function must be reset. If a signal request is already set when this function is called, the error `EOS_DEVBSY` is returned.
Release the Device

When a process no longer needs the signal installed with _os_ss_sendsig(), the process should call the function _os_ss_releas(). This function releases the device from any _os_ss_sendsig() request made by the calling process.
Playing and Recording Sound Data

Several functions are available to play and record sound data:

- **_os_ss_snd_play()**
  Begins playing from a sound map. If the hardware is not capable of playing sound, such as a record-only device, the play function returns the error EOS_UNKSVC (unknown service request).

- **_os_ss_snd_record()**
  Begins recording to a sound map. If the hardware is not capable of recording sound, such as a play-only device, the function returns the error EOS_UNKSVC (unknown service request).

- **_os_ss_snd_gain()**
  Controls the output signal level of the sound decoder.

- **_os_ss_snd_pause()**
  Pauses the play operation.

- **_os_ss_snd_cont()**
  Continues the play operation from the point at which it was paused.

- **_os_ss_snd_abort()**
  Stops the play operation.

**Play**

The play command **_os_ss_snd_play()** sends a request to the sound decoder to begin playing from a sound map. The sound data and all parameters the decoder needs to decode the data are contained in the sound map. This function requires the path ID to the sound device (established when the device was opened), a pointer to the sound map, and the blocking type.

When the first sound map is accepted by the driver, the driver sets the play and busy triggers, clears the loop counter, and sets the current offset to a value equal to the loop start value. If multiple sound maps are linked together, these fields are modified in the subsequent sound maps.
when the driver begins processing each of these subsequent sound maps. This enables applications to modify the next pointers in the sound map even after calling the play function.

The blocking type determines how the play call functions. Blocking types are described in the data structure `SND_BLOCK_TYPE`. The play function may specify one of three blocking types:

- **SND_NOBLOCK**: If the driver is busy, it immediately returns the error `EOS_MAUI_BUSY`. Otherwise, it starts the play and returns immediately.

- **SND_BLOCK_START**: The play call waits while the device is busy and returns immediately when the sound device begins to play.

- **SND_BLOCK_FINISH**: The play call waits to return until all linked sound maps have been consumed or the play operation is aborted.

If a fatal signal is received while the call is blocked waiting for access to the sound device (the play has not started yet and `block_type` is equal to `SND_BLOCK_START` or `SND_BLOCK_FINISH`), the call returns `EOS_SIGNAL` and the play request is cancelled. If a fatal signal is received after the play has started and the call is blocked waiting for the play to finish (`block_type` is equal to `SND_BLOCK_FINISH`), the call returns `EOS_SIGNAL`, but the play operation continues. Examine `smap->trig_status` to determine if the sound map is still being used by the driver.
When the sound decoder completes an activity that satisfies a trigger in the sound map trigger mask and the sound map trigger signal is not set to zero, the driver sends the signal `smap->trig_signal` to the calling process. The following figure shows the status and trigger points of a typical play operation.

**Figure 2-1** Play Operation Status and Trigger Points.

![Diagram showing play operation status and trigger points]

**Note**

Not all drivers support all triggers. Check the device capabilities to determine which triggers are supported by your hardware.

**Record**

The record command `_os_ss_snd_record()` sends a request to the sound encoder to begin recording to a sound map. The sound data and all parameters the encoder needs to encode the data are contained in the sound map. This function requires the path ID to the sound device (established when the device was opened), a pointer to the sound map, and the block type.
When the first sound map is accepted by the driver, the driver sets the record and busy triggers. If multiple sound maps are linked together, these fields are modified in the subsequent sound maps when the driver begins processing each of these subsequent sound maps. This enables applications to modify the next pointers in the sound map even after calling the record function.

Record operations do not support loop operations. The SND_SMAP fields loop_start, loop_end, loop_count, and loop_counter must be set to zero by the application before the sound map is accepted by the driver.

The blocking type determines how the record call functions. Blocking types are described in the data structure SND_BLOCK_TYPE. The record function may specify one of three blocking types:

- **SND_NOBLOCK**: If the driver is busy, it immediately returns the error EOS_MAUI_BUSY. Otherwise, it starts the record and returns immediately.

- **SND_BLOCK_START**: The record call waits while the device is busy and returns immediately when the sound device begins to record.

- **SND_BLOCK_FINISH**: The record call waits to return until all linked sound maps have been consumed or the record operation is aborted.

If a fatal signal is received while the call is blocked waiting for access to the sound device (the record has not started yet and block_type is equal to SND_BLOCK_START or SND_BLOCK_FINISH), the call returns EOS_SIGNAL and the record request is cancelled. If a fatal signal is received after the record has started and the call is blocked waiting for the record to finish (block_type is equal to SND_BLOCK_FINISH), the call returns EOS_SIGNAL, but the record operation continues. Examine the smap->trig_status to determine if the sound map is still being used by the driver.
When the sound encoder completes an activity that satisfies a trigger in the sound map trigger mask and the sound map trigger signal is not set to zero, the driver sends the signal `smap->trig_signal` to the calling process. The following figure shows the status and trigger points of a typical recording operation.

**Figure 2-2 Record Operation Status and Trigger Points**

![Record Operation Status and Trigger Points Diagram](image)

**Note**
Not all drivers support all triggers. Check the device capabilities to determine which triggers are supported by your hardware.

**Gain Control**

The output signal gain for play operations and the input signal gain for record operations is controlled with the function `os_ss_snd_gain()`. This function requires the path ID of the sound device, and specifies the mix lines and the gain operation to perform.
The gain capabilities of the sound device are stored in a data structure named `SND_GAIN_CAP`. This data structure is returned as part of the `SND_DEV_CAP` structure when the application calls `_os_gs_snd_devcap()`. The `SND_GAIN_CAP` includes the following capabilities information:

**Table 2-1  SND_GAIN_CAP Structure**

<table>
<thead>
<tr>
<th>Types</th>
<th>Supported Gain Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>Mask of one or more mix lines that share a <code>SND_GAIN_CAP</code> description.</td>
</tr>
<tr>
<td>Support for Mute</td>
<td>Boolean <code>TRUE</code> if muting is supported.</td>
</tr>
<tr>
<td>Default Type</td>
<td>The driver's default gain type for the mix lines.</td>
</tr>
<tr>
<td>Default Level</td>
<td>The driver's initial gain value for the indicated mix lines.</td>
</tr>
<tr>
<td>Zero Level</td>
<td>Level value where the change in dB is zero.</td>
</tr>
<tr>
<td>Number of Step</td>
<td>Number of actual gain or attenuation values supported by the hardware.</td>
</tr>
<tr>
<td>Step size</td>
<td>Average size of each step of gain, expressed in 100th of a dB.</td>
</tr>
<tr>
<td>Minimum dB</td>
<td>Change in dB at level <code>SND_GAIN_MIN</code>.</td>
</tr>
<tr>
<td>Maximum dB</td>
<td>Change in dB at level <code>SND_GAIN_MAX</code>.</td>
</tr>
</tbody>
</table>

Gain is controlled by increasing or decreasing gain in increments. The step size gives you information about the average size of an increment. However, the incremental increase or decrease may not be linear. The zero level is the level at which all increases are gain and all decreases are attenuation. Zero gain does not imply zero dB.
Levels above zero are represented by positive gain values, and levels below the zero level are negative gain values. The gain levels are absolute levels, not relative to the current setting.

For example, if the zero setting correlates to a sound output that is 48 dB, and step size is 6.00 dB, increasing the play gain level to 2 increases the volume of the sound output to roughly 60 dB. To set the volume back to 48 dB, you set the play gain to zero. Setting the gain to -2 decreases the volume to 36 dB which is two steps below the 48 dB zero-level. The following figure illustrates how play gain works.

**Figure 2-3  Effects of Play Gain Level on Volume**

(Note)

The hardware specification for the particular driver that is installed provides more specific information regarding gain capabilities of the hardware. Contact the hardware manufacturer for the hardware specification for your driver.)
The function `_os_ss_snd_gain()` sends a request to the sound driver to modify one or more mix lines. The following list describes each of the operations that can be applied with the `_os_ss_snd_gain()` function:

- **SND_GAIN_CMD_UP**: Specifies the number of steps to increment the gain level of the mix line. This has the effect of increasing the volume or input signal level.
- **SND_GAIN_CMD_DOWN**: Specifies the number of steps to decrement the gain level of the mix line. This has the effect of decreasing the volume or input signal level.
- **SND_GAIN_CMD_RESET**: Indicates to reset the gain level to the default level found in `SND_GAIN_CAP`.
- **SND_GAIN_CMD_MUTE**: Indicates to set or clear the mute bit according to state in `SND_GAIN_MUTE`.
- **SND_GAIN_CMD_MONO**: Sets a specific mono gain level.
- **SND_GAIN_CMD_STEREO**: Sets the specific left and right gain levels for stereo.
- **SND_GAIN_TYPE_XSTEREO**: Sets a specific gain level for the X-stereo.

Mono is a single channel output. Stereo is a two-channel (left and right) output. X-stereo is a two-channel output that has the additional capability of switching the left channel to the right output and the right channel to the left output.
Sound Operations

Pause

The pause command `os_ss_snd_pause()` sends a request to the sound driver to pause the active operation. When the operation is paused, the `SND_STATUS_PLAY_PAUSED` or `SND_STATUS_RECORD_PAUSED` bit in the status field of `SND_DEV_STATUS` is set. Only the path that initiated the operation can pause the operation. If another path attempts to pause an operation, the error `EOS_MAUI_NOTOWNER` is returned. If a path is opened for both read and write, the pause command pauses the play and record operations that are running.

If the pause command is not supported by the hardware, this function returns the error `EOS_MAUI_NOHWSUPPORT` (no hardware support) and the driver continues to play.

Continue

The continue command `os_ss_snd_cont()` sends a request to the sound driver to continue a currently paused operation. This function clears the `SND_STATUS_PLAY_PAUSED` or `SND_STATUS_RECORD_PAUSED` bit in the status field of `SND_DEV_STATUS`, then begins playing the sound data at the point at which it was paused.

Only the path that initiated the operation and issued the pause command can continue the operation. If another path attempts to continue a paused operation, the error `EOS_MAUI_NOTOWNER` is returned.

The `sup_play_pause` and `sup_record_pause` fields of `SND_DEV_CAP` indicate if the sound driver supports pause and continue commands. If the command is not supported, this function returns the error `EOS_UNKSVC`. 
Abort

The abort command _os_ss_snd_abort() sends a request to the sound driver to synchronously abort the current play or record operation.

Only the path that initiated the operation can abort the operation. If another path attempts to abort an operation, the error EOS_MAUI_NOTOWNER is returned.
Completing Sound Operations

One entry point is available for completing sound operations.

_os_close() Closes a sound device.

Close

The entry point _os_close() closes the sound device to the process that calls the function. When a process has completed all sound operations, this function should be called.
Chapter 3: Function Reference

This chapter describes each sound driver function.
Function Reference

Include Files

All applications using the Sound Driver Interface must contain the following code:

```c
#include <MAUI/mfm_snd.h>
```

Standard Driver Entry Points

From `os_lib.l`:
- `_os_open()` — Open Sound Device
- `_os_close()` — Close Sound Device
- `_os_ss_sendsig()` — Send Signal Request
- `_os_ss_relea()` — Release Signal Request

Sound Input and Output

From `mfm.l`:
- `_os_ss_snd_play()` — Play Sound to Sound Decoder
- `_os_ss_snd_record()` — Record Sound from Sound Encoder
- `_os_ss_snd_pause()` — Pause Active Sound Operation
- `_os_ss_snd_cont()` — Continue Active Sound Operation
- `_os_ss_snd_abort()` — Abort Active Sound Operation
- `_os_ss_snd_gain()` — Gain/Mixing Control
Device Compatibility, Capability, and Status

From mfm.l:

_os_gs_snd_compat()  Exchange Compatibility Level
_os_gs_snd_devcap()   Get Sound Device Capabilities
_os_gs_snd_status()   Get Sound Device Status
Function Reference

_os_close()
Close Path to Sound Device

Syntax

```c
error_code
_os_close(path_id path)
```

Parameter Block

typedef struct i_close_pb {
    syscb cb;
    path_id path;
} i_close_pb, *I_close_pb;

Description

_os_close() is fully documented in Ultra C Library Reference.

path specifies an open path ID to the sound device.

Direct Errors

- EOS_BPNUM Bad path number.

See Also

_os_open()

_os_close in the Ultra C Library Reference
I_CLOSE in the OS-9000 Technical Manual
Function Reference

_Os_gs_snd_compat()

Exchange Compatibility Level

Syntax

```
error_code _os_gs_snd_compat(path_id path, u_int32 *ret_sdv_compat, u_int32 api_compat)
```

Parameter Block

```
typedef struct {
    u_int16 func_code; /* Must be FC_SND_COMPAT */
    u_int32 *ret_dev_compat; /* Driver compat level */
    u_int32 api_compat; /* API compat level */
} gs_snd_compat_pb, *Gs_snd_compat_pb;
```

Description

_Os_gs_snd_compat() exchanges the compatibility level of the caller with the compatibility level of the sound driver.

param specifies an open path ID to the sound device.

The compatibility level of the driver must be set by assigning the value MAUI_COMPAT_LEVEL to the return parameter ret_sdv_compat.

ret_sdv_compat is the compatibility level of the caller and should be set to MAUI_COMPAT_LEVEL. If the compatibility level of the caller is less than the driver level, then the driver must cope with the differences in the respective levels, or it must return EOS_MAUI_INCOMPATVER if it cannot cope.

If successful, this function returns SUCCESS. Otherwise, the returned value is an error code. Error codes unique to the driver are defined below.

Direct Errors

- **EOS_BPNUM** 
  Bad path number.
- **EOS_MAUI_BADPTR** 
  ret_sdv_compat is NULL.
EOS_MAUI_INCOMPATVER  The compatibility level of the caller is less than the driver and the driver cannot cope with the differences.

See Also
MAUI_COMPAT_LEVEL (See MAUI Programming Reference Manual)
Function Reference

_os_gs_snd_devcap()
Get Sound Device Capabilities

Syntax

```c
error_code _os_gs_snd_devcap(path_id path, SND_DEV_CAP **ret_dev_cap)
```

Parameter Block

typedef struct {
  u_int16 func_code; /* Must be FC_SND_DEVCAP */
  SND_DEV_CAP **ret_dev_cap;
    /* Device capabilities */
} gs_snd_devcap_pb, *Gs_snd_devcap_pb;

Description

=os_gs_snd_devcap() gets information about the capabilities of the sound device. This information may be used to adjust the operation of the application so that it runs properly on different hardware platforms.

path specifies an open path ID to the sound device.

A pointer to the buffer containing the device capabilities structure is returned in ret_dev_cap. A pointer to this variable should be passed to _os_gs_snd_devcap().

Note

Because the memory pointed to by ret_dev_cap belongs to the driver, do not attempt to modify or free this memory. Also, do not attempt to access this memory after the path is closed.

If successful, this function returns SUCCESS. Otherwise, the returned value is an error code. Error codes unique to the driver are defined as follows.
3 Function Reference

Direct Errors

EOS_BPNUM
Bad path number.

EOS_MAUI_BADPTR
ret_sdv_compat is NULL.

See Also

SND_DEV_CAP
_os_gs_snd_status()
Get Sound Device Status

Syntax

```c
error_code
_os_gs_snd_status(path_id path, SND_DEV_STATUS **ret_status)
```

Parameter Block

```c
typedef struct {
    u_int16 func_code; /* Must be FC_SND_DEV_STATUS */
    SND_DEV_STATUS **ret_status;
    /* Status of device*/
} gs_snd_status_pb, *Gs_snd_status_pb;
```

Description

_os_gs_snd_status() returns the current status of the sound hardware. This function may be called by any process at any time.

`path` specifies an open path ID to the sound device. The SND_DEV_STATUS for a device is unique for each logical unit, not necessarily unique per `path`.

A pointer to the buffer containing the device logical unit status structure is returned in `ret_status`. A pointer to this variable should be passed to _os_gs_snd_status().

Note

Because the memory pointed to by `ret_status` belongs to the driver, do not attempt to modify or free this memory. Also do not attempt to access this memory after the `path` is closed.

If successful, this function returns `SUCCESS`. Otherwise, the returned value is an error code. Error codes unique to the driver are defined below.
**Direct Errors**

EOS_BPNUM  
Bad path number.

EOS_MAUI_BADPTR  
ret_sdv_compat is NULL.

**See Also**

_os_open()  
SND_DEV_STATUS
Function Reference

**_os_open()**

Open Path to Sound Device

**Syntax**

```c
#include <modes.h>
error_code
_os_open(char *name, u_int32 mode, path_id *path)
```

**Parameter Block**

```c
#include <srvcb.h>
typedef struct i_open_pb {
  syscb cb;
  u_char *name;
  u_int16 mode;
  path_id path;  /* output */
} i_open_pb, *I_open_pb;
```

**Description**

_os_open() is fully documented in *Ultra C Library Reference*.  

name is a pointer to the path name of the sound device. On CDB equipped systems, use CDB_TYPE_SOUND in cdb_get_ddr(), to determine the name of the sound device.

mode must be set to either S_IWRITE to use _os_ss_snd_play(), or S_IREAD to use _os_ss_snd_record(). To prevent functions such as _os_ss_snd_abort(), _os_ss_snd_cont(), and _os_ss_snd_pause() from being ambiguous, both S_IWRITE and S_IREAD must not be set on the same path. To perform both plays and records on the same device, simply open another path to the same device with the appropriate mode. Only 16 bits of mode are used.

path is a pointer to the location where _os_open() stores the resulting path number. Multiple paths may be open to the same device. Use _os_close to return the resources to the system.
If successful, this function returns `SUCCESS`. Otherwise, the returned value is an error code. Error codes unique to the driver are defined below. This call uses the standard `_os_open()` binding found in `os_lib.l`.

**Direct Errors**
- **EOS_BMODE**: Bad I/O Mode.
- **EOS_BPNUM**: Bad path number.
- **EOS_FNA**: File not accessible.
- **EOS_PNNF**: Path name not found.
- **EOS_PTHFUL**: The user's (or system) path table is full.
- **EOS_SHARE**: Non-sharable file/device is busy.

**See Also**
- `_os_close()`
- `_os_ss_snd_abort()`
- `_os_ss_snd_cont()`
- `cdb_get_ddr()`
- `CDB_TYPE_SOUND`

**MAUI Programming Reference Manual**
- `_os_open`
- `I_OPEN, S_IREAD, S_IWRITE`

**Ultra C Library Reference**
- `OS-9000 Technical Manual`
Function Reference

_os_ss_relea()

Release Signal Request

Syntax
#include <sg_codes.h>
error_code
_os_ss_relea(path_id path)

Parameter Block
#include <srvcb.h>
typedef struct f_clrsigs_pb {
    syscb cb;
    process_id proc_id;
} f_clrsigs_pb, *F_clrsigs_pb;

Description
_os_ss_relea() releases the device from any _os_ss_sendsig() request made by the calling process.
path is the path number of the device to release.
If no signal request is active, SUCCESS is returned.
If successful, this function returns SUCCESS. Otherwise, the returned value is an error code. Error codes unique to the driver are defined below.
This call uses the standard _os_ss_relea() binding found in os_lib.l.

Direct Errors
EOS_BPNUM Bad path number.
EOS_PERMIT Caller is not the path that installed the signal request.

See Also
_os_ss_sendsig()
Function Reference

_os_ss_sendsig()
Send Signal Request

Syntax
#include <sg_codes.h>
error_code
_os_ss_sendsig(path_id path, signal_code signal)

Parameter Block
#include <srvcb.h>
typedef struct f_clrsigs_pb {
    syscb cb;
    process_id proc_id;
    signal_code signal;
} f_clrsigs_pb, *F_clrsigs_pb;

Description
_os_ss_sendsig() sets up a signal to send to a process when the sound device is ready to accept a new sound operation such as _os_ss_snd_play() and _os_ss_snd_record(). This function is useful to determine when the sound device becomes idle. _os_ss_sendsig() must be reset each time the signal is sent.

For example, process A is playing sound data. Process B could issue an _os_ss_sendsig() to receive a signal when it can submit it’s sound data without blocking.

path specifies an open path ID to the sound device.

signal specifies the signal to send when the driver clears the SND_STATUS_BUSY bit in the device status (see _os_gs_snd_status() on page 47). If the device is ready when _os_ss_sendsig() is called, signal is sent immediately.

Receipt of signal by a process is not a guarantee that the process will get access to the device. Another process could take control of the device between when the signal is sent, and when the receiving process attempts to use the device.
If successful, this function returns SUCCESS. Otherwise, the returned value is an error code. Error codes unique to the driver are defined below.

This call uses the standard _os_ss_sendsig() binding found in os_lib.l.

**Direct Errors**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOS_BPNUM</td>
<td>Bad path number.</td>
</tr>
<tr>
<td>EOS_DEVBSY</td>
<td>Signal request is already set.</td>
</tr>
</tbody>
</table>

**See Also**

- _os_gs_snd_status()
- _os_ss_relea()
- SND_DEV_STATUS
- SND_STATUS
Abort Active Sound Operation

**Syntax**

```c
error_code
_os_ss_snd_abort(path_id path)
```

**Parameter Block**

```c
typedef struct {
    u_int16 func_code; /* Must be FC_SND_ABORT */
} ss_snd_abort_pb, *Ss_snd_abort_pb;
```

**Description**

>_os_ss_snd_abort()_ requests the driver to synchronously abort the currently active sound operation such as _os_ss_snd_play() and _os_ss_snd_record(). If there is no sound operation active, this function returns EOS_MAUI_NOTBUSY.

*path* specifies an open path ID to the sound device.

If a sound operation is currently active, it is aborted, EOS_ABORT is placed in the err_code member of the SND_SMAP referenced by the operation and the functions returns when the device is ready for the next sound operation.

The SND_STATUS_FINISH bit in the trig_status member of SND_SMAP is set and the signal trig_signal specified in SND_SMAP is sent to the calling process. If the specified signal is zero, then no signal is sent.

If this function is called with a *path* other than the *path* that initiated the sound operation, this function returns EOS_MAUI_NOTOWNER. If successful, this function returns SUCCESS.

**Direct Errors**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOS_BPNUM</td>
<td>Bad path number.</td>
</tr>
<tr>
<td>EOS_MAUI_NOTBUSY</td>
<td>The sound hardware is not busy playing or recording, so there is nothing to abort.</td>
</tr>
</tbody>
</table>
EOS_MAUI_NOTOWNER

Caller is not the path that started the current sound operation.

See Also

_os_ss_snd_cont()
_os_ss_snd_pause()
_os_ss_snd_play()
_os_ss_snd_record()

SND_SMAP
Using the Sound Driver Interface

_function Reference_

_os_ss_snd_cont()

Continue Active Sound Operation

**Syntax**

```
error_code _os_ss_snd_cont(path_id path)
```

**Parameter Block**

```
typedef struct {
    u_int16 func_code;/* Must be FC_SND_CONT */
} ss_snd_cont_pb, *Ss_snd_cont_pb;
```

**Description**

_os_ss_snd_cont() requests the driver to continue the current paused sound operation such as _os_ss_snd_play() or _os_ss_snd_record(). If there is no sound operation paused, this function returns EOS_MAUI_NOTPAUSED.

*path* specifies an open path ID to the sound device.

If a play is currently paused, it is continued and the SND_STATUS_PLAY_PAUSED bit in status of SND_DEV_STATUS is cleared. If a record is currently paused, it is continued and the SND_STATUS_RECORD_PAUSED bit in status of SND_DEV_STATUS is cleared. No other status bits are affected by the pause.

If the hardware does not support pause or continue, _os_ss_snd_cont() returns EOS_MAUI_NOHWSUPPORT.

If this function is called with a *path* other than the *path* that initiated the sound operation, this function returns EOS_MAUI_NOTOWNER. If successful, this function returns SUCCESS.

**Direct Errors**

- **EOS_BPNUM**
  - Bad path number.
- **EOS_MAUI_NOHWSUPPORT**
  - The hardware does not support pause and continue.
Function Reference

EOS_MAUI_NOTBUSY

The sound hardware is not busy so there is no play or record to act on.

EOS_MAUI_NOTOWNER

Caller is not the path that started the current sound operation.

EOS_MAUI_NOTPAUSED

The sound hardware is not paused so there is no play or record to act on.

See Also
_
_os_ss_snd_abort()
_os_ss_snd_pause()
_os_ss_snd_play()
_os_ss_snd_record()
SND_SMAP
_os_ss_snd_gain()

Set Gain

Syntax

error_code
=os_ss_snd_gain(path_id path, SND_GAIN *gain)

Parameter Block

typedef struct {
   u_int16 func_code;/* Must be FC_SND_GAIN */
   SND_GAIN *gain;/* Gain Setting */
} ss_snd_gain_pb, *Ss_snd_gain_pb;

Description

_os_ss_snd_gain() requests the driver to change the gain level.
path specifies an open path ID to the sound device.
gain is a pointer to a data structure that describes how to change the
gain level and what lines are affected. (See SND_GAIN on page 92.) If
this function is called with a path other than the path that initiated the
current sound operation, this function returns EOS_MAUI_NOTOWNER.
_os_gs_snd_devcap() may be used to determine which mix lines
are supported by the hardware. _os_gs_snd_status() may be used
to determine current gain levels.
If successful, this function returns SUCCESS.

Direct Errors

EOS_BPNUM           Bad path number.
EOS_MAUI_BADPTR     The pointer gain is NULL.
EOS_MAUI_BADVALUE   Unknown gain command in
gain->cmd.
EOS_MAUI_NOHWSUPPORT The value passed in play->cmd is not supported by the hardware.
Function Reference

**EOS_MAUI_NOTOWNER**

Caller is not the path that started the current sound operation.

**EOS_UNKSVC**

The hardware is incapable of supporting gain control for the specified line.

**See Also**

-_os_gs_snd_devcap()_
-_os_gs_snd_status()_
*SND_DEV_CAP*
*SND_DEV_STATUS*
*SND_GAIN*
Function Reference

_os_ss_snd_pause()

Pause Active Sound Operation

Syntax

```c
error_code _os_ss_snd_pause(path_id path)
```

Parameter Block

```c
typedef struct {
    u_int16 func_code; /* Must be FC_SND_PAUSE */
} ss_snd_pause_pb, *Ss_snd_pause_pb;
```

Description

_os_ss_snd_pause() requests the driver to pause the current active sound operation such as _os_ss_snd_play() or _os_ss_snd_record(). If there is no sound operation active, this function returns EOS_MAUI_NOTBUSY.

path specifies an open path ID to the sound device.

If a play is currently active, it is paused and the SND_STATUS_PLAY_PAUSED bit in status of SND_DEV_STATUS is set. If a record is currently active, it is paused and the SND_STATUS_RECORD_PAUSED bit in status of SND_DEV_STATUS is set. No other status bits are affected by the pause.

If the hardware does not support pause and continue, sup_play_pause returns EOS_MAUI_NOHWSUPPORT.

If this function is called with a path other than the path that initiated the sound operation, this function returns EOS_MAUI_NOTOWNER.

If successful, this function returns SUCCESS. Otherwise, the returned value is an error code. Error codes unique to the driver are defined below.

Direct Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOS_BPNUM</td>
<td>Bad path number.</td>
</tr>
</tbody>
</table>
Function Reference

EOS_MAUI_NOHWSUPPORT

The hardware does not support pause and continue.

EOS_MAUI_NOTOWNER

Caller is not the path that started the current sound operation.

EOS_MAUI_NOTBUSY

The sound hardware is not busy playing or recording so there is no play or record to act on.

EOS_MAUI_PAUSED

The sound hardware is already paused.

See Also

_os_ss_snd_abort()
_os_ss_snd_cont()
_os_ss_snd_play()
_os_ss_snd_record()
SND_DEV_CAP
SND_DEV_STATUS
SND_SMAP
Function Reference

_os_ss_snd_play()
Play Sound to Sound Decoder

Syntax

error_code
_os_ss_snd_play(path_id path, SND_SMAP *smap,
SND_BLOCK_TYPE block_type)

Parameter Block
typedef struct {
   u_int16 func_code;/* Must be FC_SND_PLAY */
   SND_SMAP *smap;/* Sound map */
   SND_BLOCK_TYPE block_type;
   /* Type of blocking to */
   /* perform */
} ss_snd_play_pb, *Ss_snd_play_pb;

Description
_os_ss_snd_play() requests the sound decoder to decode and play sound data. The sound data and all parameters necessary to decode it, are stored in the sound map referenced by the pointer smap.

path specifies an open path ID to the sound device. path must be opened for write access.

The driver only processes one SND_SMAP at a time, following the smap->next pointers as the sound data is consumed. As each SND_SMAP is accepted by the driver, it sets the SND_TRIG_BUSY bit in smap->trig_status, clears smap->loop_counter, and sets smap->cur_offset equal to smap->loop_start.

The blocking type determines how the driver functions. One of three blocking types must be specified:
### Function Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SND_NOBLOCK</td>
<td>prevents the driver from blocking. If the driver is busy when called, it immediately returns the error EOS_MAUI_BUSY. Otherwise, it starts the play and returns immediately.</td>
</tr>
<tr>
<td>SND_BLOCK_START</td>
<td>returns when the play starts (when SND_TRIG_START is set). The play call waits while the device is busy and returns immediately when the sound device begins to play.</td>
</tr>
<tr>
<td>SND_BLOCK_FINISH</td>
<td>returns when the play is finished (when SND_TRIG_FINISHED is set). The play call waits to return until all linked sound maps are consumed or the play operation is aborted.</td>
</tr>
</tbody>
</table>

If a fatal signal is received while the call is blocked waiting for access to the sound device (the play has not started yet and block_type is equal to SND_BLOCK_START or SND_BLOCK_FINISH) the call returns EOS_SIGNAL and the play request is canceled. If a fatal signal is received after the play has started, and the call is blocked waiting for the play to finish (block_type is equal to SND_BLOCK_FINISH) the call returns EOS_SIGNAL, but the play operation continues. Examine smap->trig_status to determine if the smap is still being used by the driver.

When the sound decoder completes an activity that satisfies a trigger in the smap->trig_mask and smap->trig_signal is not zero, the driver sends the signal smap->trig_signal to the calling process.
The following figure shows the status and trigger points of a typical play operation.

**Figure 3-1  os_ss_snd_play() Status and Trigger Points**

After consuming the data in `smap->buf`, if the `smap->next` field is not NULL, the driver sets the internal `smap` pointer to `smap->next` and continues.

If successful, this function returns `SUCCESS`. Otherwise, the returned value is an error code. Error codes unique to the driver are defined below.

**Direct Errors**

- **EOS_BMODE**: The path is not open for write access.
- **EOS_BPNUM**: Bad path number.
- **EOS_MAUI_BADNUMCHAN**: The specified `smap->num_channels` is not supported by the hardware.
- **EOS_MAUI_BADPTR**: The pointer for `smap` or `smap->buf` is NULL.
- **EOS_MAUI_BADRATE**: The specified `smap->sample_rate` is not supported by the hardware.
- **EOS_MAUI_BADSIZE**: The specified `smap->sample_size` is not appropriate for the `smap->coding_method` or is not supported by the hardware.
Using the Sound Driver Interface

EOS_MAUI_BADVALUE  The value passed for `block_type` is invalid or one or more of the loop fields are invalid*.

EOS_MAUI_BUSY  The sound decoder is busy.

EOS_MAUI_NOHWSUPPORT  The specified `smap->coding_method` is not supported by the hardware.

EOS_SIGNAL  A fatal signal was received while blocked.

EOS_UNKSVC  This function is not supported because the sound hardware does not contain a decoder.

* if (smap->loop_count > 0) {
  if (smap->loop_start >= smap->loop_end
      || smap->loop_end > smap->buf_size)
    return (smap->err_code = EOS_MAUI_BADVALUE);
}

Indirect Errors

_os_ev_anycle()
_os_send()
_os_ev_setand()
_os_ev_tstset()  OS-9 Technical Reference

See Also

_os_ss_snd_abort()
_os_ss_snd_gain()
_os_ss_snd_cont()
_os_ss_snd_pause()

SND_BLOCK_TYPE
SND_SMAP
SND_STATUS
SND_TRIGGER
Function Reference

_os_ss_snd_record()

Record Sound

Syntax

error_code
_os_ss_snd_record(path_id path, SND_SMAP *smap, SND_BLOCK_TYPE block_type)

Parameter Block

typedef struct {
    u_int16 func_code; /* Must be FC_SND_RECORD */
    SND_SMAP *smap; /* Sound map */
    SND_BLOCK_TYPE block_type; /* Type of blocking to perform */
} ss_snd_record_pb, *Ss_snd_record_pb;

Description

_os_ss_snd_record() requests the sound encoder to record sound data. The buffer for encoded sound data and all parameters necessary to encode it, are stored in the sound map referenced by the pointer smap.

path specifies an open path ID to the sound device. path must be opened for read access.

The driver only processes one SND_SMAP at a time, following the smap->next pointers as the sound data fills each smap->buf. As each SND_SMAP is accepted by the driver, it sets the SND_TRIG_BUSY bit in smap->trig_status.

_os_ss_snd_record() does not support loop operations. The SND_SMAP fields loop_start, loop_end, loop_count, and loop_counter must be set to zero by the application before the sound map is accepted by the driver. The driver returns EOS_MAUI_BADVALUE if any of these fields are not zero.

The blocking type determines how the driver functions. One of three blocking types must be specified:
### Function Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SND_NOBLOCK</td>
<td>Prevents the driver from blocking. If the driver is busy when called, it immediately returns the error EOS_MAUI_BUSY. Otherwise, it starts the record and returns immediately.</td>
</tr>
<tr>
<td>SND_BLOCK_START</td>
<td>Returns when the recording starts (when SND_TRIG_STAR is set). The record call waits while the device is busy and returns immediately when the sound device begins to record.</td>
</tr>
<tr>
<td>SND_BLOCK_FINISH</td>
<td>Returns when the record has finished (when SND_TRIG_FINISH is set). The record call waits to return until all linked sound maps are consumed or the record operation is aborted.</td>
</tr>
</tbody>
</table>

If a fatal signal is received while the call is blocked waiting for access to the sound device (the record has not started yet and block_type is equal to SND_BLOCK_START or SND_BLOCK_FINISH) the call returns EOS_SIGNAL and the record request is canceled. If a fatal signal is received, after the record has started, and the call is blocked waiting for the record to finish (block_type is equal to SND_BLOCK_FINISH) the call returns EOS_SIGNAL, but the record operation continues. Examine smap->trig_status to determine if the smap is still being used by the driver.

When the sound encoder completes an activity that satisfies a trigger in the smap->trig_mask, and smap->trig_signal is not zero, the driver sends the signal smap->trig_signal to the calling process.
The following figure illustrates the status and trigger points of a typical record operation.

**Figure 3-2  _os_ss_snd_record() Status and Trigger Points**

After filling the data in `smap->buf`, if the `smap->next` field is not NULL, the driver sets the internal `smap` pointer to `smap->next` and continues.

If successful, this function returns `SUCCESS`. Otherwise, the returned value is an error code. Error codes unique to the driver are defined below.

**Direct Errors**

- **EOS_BMODE**: The path is not open for read access.
- **EOS_BPNUM**: Bad path number.
- **EOS_MAUI_BADNUMCHAN**: The specified `smap->num_channels` is not supported by the hardware.
- **EOS_MAUI_BADPTR**: The pointer for `smap` or `smap->buf` is NULL.
- **EOS_MAUI_BADRATE**: The specified `smap->sample_rate` is not supported by the hardware.
- **EOS_MAUI_BADSIZE**: The specified `smap->sample_size` is not appropriate for the `smap->coding_method` or is not supported by the hardware.
EOS_MAUI_BADVALUE

The value passed for block_type is invalid,

or

one or more of the following smap entries are not equal to zero:

loop_start, loop_end,

loop_count, loop_counter.

EOS_MAUI_BUSY

The sound encoder is busy.

EOS_MAUI_NOHWSUPPORT

The coding_method is not supported by the hardware.

EOS_SIGNAL

A fatal signal was received while blocked.

EOS_UNKSVC

This function is not supported because the sound hardware does not contain an encoder.

Indirect Errors

_os_ev_anyclr()
_os_send()
_os_ev_setand()
_os_ev_tstset()

OS-9 Reference Manual

See Also

_os_ss_snd_abort()
_os_ss_snd_cont()
_os_ss_snd_gain()
_os_ss_snd_pause()

SND_BLOCK_TYPE
SND_SMAP
SND_STATUS
SND_TRIGGER
Chapter 4: Data Type Reference

This chapter provides a detailed reference for each data type defined in the Sound Driver Interface.
Data Type Reference

Defined Constants

SND_LEVEL_*  Gain Level Constants

Enumerated Types

SND_BLOCK_TYPE  Blocking Types

Data Types

SND_CM  Sound Coding Methods

Integers

SND_GAIN_CMD  Gain Commands
SND_LINE  Gain/Mixer Line Types
SND_STATUS  Status
SND_TRIGGER  Triggers

Data Structures

SND_DEV_CAP  Sound Device Capabilities
SND_DEV_CM  Sound Device Coding Method
SND_DEV_STATUS  Sound Device Status
SND_GAIN  Gain Control
SND_GAIN_UP  Gain Up Parameters
<table>
<thead>
<tr>
<th>Data Type Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SND_GAIN_DOWN</td>
<td>Gain Down Parameters</td>
</tr>
<tr>
<td>SND_GAIN_MUTE</td>
<td>Mute Gain Parameters</td>
</tr>
<tr>
<td>SND_GAIN_MONO</td>
<td>Mono Gain Parameters</td>
</tr>
<tr>
<td>SND_GAIN_STEREO</td>
<td>Stereo Gain Parameters</td>
</tr>
<tr>
<td>SND_GAIN_XSTEREO</td>
<td>Cross-Stereo Gain Parameters</td>
</tr>
<tr>
<td>SND_GAIN_CAP</td>
<td>Sound Device Gain Capabilities</td>
</tr>
<tr>
<td>SND_SMAP</td>
<td>Sound Map</td>
</tr>
</tbody>
</table>
SND_BLOCK_TYPE

Blocking Types

Syntax
typedef enum {
    SND_BLOCK_START,   /* Block until sound */
                       /* operation starts */
    SND_BLOCK_FINISH,  /* Block until sound */
                       /* operation ends */
    SND_NOBLOCK        /* Do not block */
} SND_BLOCK_TYPE;

Description
This enumerated type defines the blocking mechanisms that are available when playing or recording sound samples.

The blocking type determines the behavior of the sound operation functions _os_ss_snd_play() and _os_ss_snd_record(). These functions are called with one of three blocking types:

SND_BLOCK_START returns when the sound operation starts. The function call waits while the device is busy and returns immediately when the sound device begins to play or record.

SND_BLOCK_FINISH returns when the sound operation is finished. The function call waits to return until all linked sound maps are consumed or the sound operation is aborted.

SND_NOBLOCK prevents the driver from blocking. If the driver is busy when called, it immediately returns the error EOS_MAUI_BUSY. Otherwise, it starts the sound operation and returns immediately.
See Also

_os_ss_snd_play()
_os_ss_snd_record()
**SND_CM**
Sound Coding Methods

**Syntax**

typedef u_int32 SND_CM;

**Description**

This data type specifies a sound coding method. In a SND_SMAP, the SND_CM is used to tell the driver the type of sound data being passed to it for play operations, or what is required from it during record operations.

Be aware that not all sound coding methods can be encoded or decoded by all hardware. Use _os_ss_snd_devcap() to determine which sound coding methods the driver and hardware support.

The following diagram shows the bit-fields contained in the sound coding method:

**Figure 4-1  Coding Method**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| OEM | Reserved | Reserved | Reserved | Name |

**OEM**

Bits 31 through 25 may be used by OEMs to indicate implementation- or driver-specific coding method modifiers.
**BE**

Bit 19 is a coding method modifier that indicates the byte endianess of the sound data. If this bit is set, the sound data is formatted as least significant byte first. If this bit is not set, the sound data is formatted as most significant byte first. This bit is considered a modifier to the sound coding method name.

Use the macro `snd_get_cm_byte_order(cm)` to get the byte endianess in a `SND_CM`. For example:

```c
byteorder=snd_get_cm_byte_order(cm);
```

The `byteorder` variable is equal to `MSBFIRST` or `LSBFIRST`.

Use the macro `snd_set_cm_byte_order(order)` with `order` equal to `MSBFIRST` or `LSBFIRST`, or use the macros `SND_CM_MSBYTE1ST` or `SND_CM_LSBYTE1ST` to set the byte endianess in a `SND_CM`.

The following two code segments set least significant byte ordering to the variable `cm`:

```c
cm=SND_CM_LSBYTE1ST |
    snd_get_cm_name(SND_CM_PCM_SLINEAR);
cm=snd_set_cm_byte_order(LSBFIRST) |
    snd_get_cm_name(SND_CM_PCM_SLINEAR);
```

Most significant byte ordering is the default. You do not need to use either of these macros, but they are provided for completeness. The following two code segments set most significant byte ordering to the variable `cm`:

```c
cm=SND_CM_MSBYTE1ST |
    snd_get_cm_name(SND_CM_PCM_SLINEAR);
cm=snd_set_cm_byte_order(LSBFIRST) |
    snd_get_cm_name(SND_CM_PCM_SLINEAR);
```
bE

Bit 18 is a coding method modifier that indicates the bit endianess of the sound data. If this bit is set, the sound data is formatted as least significant bit first. If this bit is not set, the sound data is formatted as most significant bit first. This bit is considered a modifier to the sound coding method name.

Use the macro `snd_get_cm_bit_order(cm)` to get the bit endianess in a `SND_CM`. For example:

```c
    bitorder=snd_get_cm_bit_order(cm);
```

The `bitorder` variable is equal to `MSBFIRST` or `LSBFIRST`.

Use the macro `snd_set_cm_bit_order(order)` with order equal to `MSBFIRST` or `LSBFIRST`, or use the macros `SND_CM_MSBIT1ST` or `SND_CM_LSBIT1ST` to set the bit endianess in a `SND_CM`.

The following two code segments set least significant bit ordering to the variable `cm`:

```c
    cm=SND_CM_LSBIT1ST | snd_get_cm_name(SND_CM_PCM_SLINEAR);
    cm=snd_set_cm_bit_order(LSBFIRST) | snd_get_cm_name(SND_CM_PCM_SLINEAR);
```

Most significant bit ordering is the default. You do not need to use either of these macros, but they are provided for completeness. The following two code segments set most significant bit ordering to the variable `cm`:

```c
    cm=SND_CM_MSBIT1ST | snd_get_cm_name(SND_CM_PCM_SLINEAR);
    cm=snd_set_cm_bit_order(LSBFIRST) | snd_get_cm_name(SND_CM_PCM_SLINEAR);
```
Name

Bits 0 through 9 define the name of the coding method. This field is segmented into the following numeric ranges to indicate the class of coding method as defined in the following table:

Table 4-1 Coding Method Classes

<table>
<thead>
<tr>
<th>Numeric Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 255</td>
<td>Standard Microware-defined coding methods.</td>
</tr>
<tr>
<td>256 - 767</td>
<td>Reserved.</td>
</tr>
<tr>
<td>768 - 1023</td>
<td>Defined by OEMs.</td>
</tr>
</tbody>
</table>

The following standard coding method names are defined:

Table 4-2 Standard Coding Method Names

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SND_CM_UNKNOWN</td>
<td>Unknown or not yet determined.</td>
</tr>
<tr>
<td>1</td>
<td>SND_CM_PCM_ULAW</td>
<td>μLAW encoded PCM.</td>
</tr>
<tr>
<td>2</td>
<td>SND_CM_PCM_ALAW</td>
<td>ALAW encoded PCM.</td>
</tr>
<tr>
<td>3</td>
<td>SND_CM_PCM_SLINEAR</td>
<td>Signed Linear encoded PCM.</td>
</tr>
<tr>
<td>4</td>
<td>SND_CM_PCM_ULINEAR</td>
<td>Unsigned Linear encoded PCM.</td>
</tr>
<tr>
<td>5</td>
<td>SND_CM_ADPCM_G721</td>
<td>CCITT G.721 ADPCM.</td>
</tr>
</tbody>
</table>
Use the macro `snd_get_cm_name(cm)` to extract the coding method name from a `SND_CM`. For example:

```c
name = snd_get_cm_name(cm);
```

Use the macro `snd_set_cm_name(name)` to set the coding method name from a `SND_CM`. For example:

```c
cm = snd_set_cm_name(name);
```

**SND_CM_UNKNOWN**

This type value indicates that the sound coding method is unknown or unspecified.

**SND_CM_PCM_SLINEAR**,  
**SND_CM_PCM_SLINEAR | SND_CM_LSBYTE1ST**  
**SND_CM_PCM_ULINEAR**  
**SND_CM_PCM_ULINEAR | SND_CM_LSBYTE1ST**

Pulse Coded Modulation (PCM) is one of the most common methods used by sound cards to record and play back recorded sound.  
**SND_CM_PCM_SLINEAR** is defined as signed (2's complement) linear PCM samples.  
**SND_CM_PCM_ULINEAR** represents unsigned PCM data.

Multi-channel samples under **SND_CM_PCM_SLINEAR** and  
**SND_CM_PCM_ULINEAR** are defined to be interleaved on a frame-by-frame basis: if there are N channels, the data is a sequence of

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>SND_CM_ADPCM_G723</td>
<td>CCITT G.723 ADPCM.</td>
</tr>
<tr>
<td>7</td>
<td>SND_CM_ADPCM_IMA</td>
<td>IMA ADPCM.</td>
</tr>
</tbody>
</table>

Table 4-2 Standard Coding Method Names
frames, where each frame contains N samples, one from each channel (thus, the sampling rate is really the number of frames per second). For stereo, the left channel comes first, followed by the right channel.

Mono and stereo 8-bit linear PCM data is depicted in the following two figures.

**Figure 4-2  Mono, 8-bit Data**

```
SND_CM_PCM_SLINEAR, SND_CM_PCM_ULINEAR,
SND_CM_PCM_ULAW, SND_CM_PCM_ALAW
```

For 16-bit or larger linear PCM data, either the `SND_CM_MSBYTE1ST` or the `SND_CM_LSBYTE1ST` modifiers may be used. Mono and stereo 16-bit `SND_CM_PCM_SLINEAR` and `SND_CM_PCM_ULINEAR` are depicted in the following two figures.
Mono and stereo 16-bit SND_CM_PCM_SLINEAR | SND_CM_PCM_ULINEAR | SND_CM_LSBYTE1ST and SND_CM_PCM_ULINEAR | SND_CM_LSBYTE1ST are depicted in the following two figures.
Figure 4-7  Mono 16-bit, Little-Endian Data

SND_CM_PCM_SLINEAR|SND_CM_LSBYTE1ST
SND_CM_PCM_ULINEAR|SND_CM_LSBYTE1ST

Time

32-bit Word

sample 6  sample 5  sample 4  sample 3  sample 2  sample 1
MONO    MONO
31  24  23  16  15  8  7  0

Figure 4-8  16-bit, Little-Endian Data

SND_CM_PCM_SLINEAR|SND_CM_LSBYTE1ST
SND_CM_PCM_ULINEAR|SND_CM_LSBYTE1ST

Time

32-bit Word

sample 3  sample 3  sample 2  sample 2  sample 1  sample 1
RIGHT    LEFT
31  24  23  16  15  8  7  0
The \( \mu \)-Law (pronounced mu-law) format is an 8-bit companded format that is a telephony standard used in the United States and Japan. The A-Law format is an 8-bit companded format that is a telephony standard used in Europe. Under these encoding methods, the samples are logarithmically encoded into 8 bits. The quantization is not uniform (as in \texttt{SND\_CM\_PCM\_SLINEAR} and \texttt{SND\_CM\_PCM\_ULINEAR}), but is skewed so that the signal is sampled with greater resolution when the amplitude is less.

The official definition for \( \mu \)-Law is contained in the CCITT standard G.711. \texttt{SND\_CM\_PCM\_ULAW} and \texttt{SND\_CM\_PCM\_ALAW} are depicted in the two following figures.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4-9.png}
\caption{Stereo 16-bit, Big-Endian Data}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4-9.png}
\caption{Stereo 8-bit, Data}
\end{figure}
SND_CM_PCM_ALAW is depicted in the following two figures:

**Figure 4-10 Stereo 16-bit, Big-Endian Data**

SND_CM_PCM_SLINEAR, SND_CM_PCM_ULINEAR

\[ \text{Time} \quad 32\text{-bit Word} \]

<table>
<thead>
<tr>
<th>sample 2</th>
<th>sample 2</th>
<th>sample 2</th>
<th>sample 2</th>
<th>sample 1</th>
<th>sample 1</th>
<th>sample 1</th>
<th>sample 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIGHT LO</td>
<td>LEFT HI</td>
<td>RIGHT LO</td>
<td>LEFT HI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 15 24 31 16 23</td>
<td>0 15 24 31 16 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4-10 Stereo 8-bit, Data**

SND_CM_PCM_SLINEAR, SND_CM_PCM_ULINEAR, SND_CM_PCM_ULAW, SND_CM_PCM_ALAW

\[ \text{Time} \quad 32\text{-bit Word} \]

<table>
<thead>
<tr>
<th>sample 3</th>
<th>sample 2</th>
<th>sample 2</th>
<th>sample 2</th>
<th>sample 1</th>
<th>sample 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIGHT</td>
<td>LEFT</td>
<td>RIGHT</td>
<td>LEFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 7 15 23 16 24</td>
<td>0 7 15 23 16 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No modifiers are appropriate for SND_CM_PCM_ULAW or SND_CM_PCM_ALAW.

**SND_CM_ADPCMIMA**

Adaptive Differential Pulse Code Modulation (ADPCM) is used for improved performance and compression ratios over μ-Law and A-Law. The IMA ADPCM format uses the DVI ADPCM algorithm. It provides a 4-to-1 compression ratio (4 bits are saved for each 16-bit sample capture).

For more detailed information on the IMA ADPCM format contact IMA at (410) 626-1380.

The SND_CM_ADPCMIMA format is depicted in the following two figures.
ADPCM stands for Adaptive Delta Pulse Code Modulation. G.721 is a CCITT standard for ADPCM at 32 kbits/second.

* There are two forms of IMA ADPCM in common use. The most common form found in WAV audio files is a WAV type 0x11. This is an enhanced definition of the IMA ADPCM specification that divides the data into autonomous blocks. This form is usually encoded/decoded in software from/to 16-bit PCM. The second form of IMA ADPCM found in WAV audio files is a WAV type 0x39. This form follows the original IMA specification and is not “blocked”. This latter form can be encoded and decoded directly by several popular CODECs. This second form is represented by the SND_CM type SND_CM_ADPCM.IMA.
**SND_CM_ADPCM_G723**

G.723 is a CCITT standard for ADPCM at 24 and 40 kbits/second.

**SND_CM_OEM_***

We recommend that if an OEM defines a coding method, that it be prefixed with `SND_CM_OEM_*` and that it be in the range 768 through 1023. We also request that OEMs submit a detailed description of their coding methods to Microware. These coding methods may become part of the Standard Coding Method Names in future releases.

**See Also**

`_os_gs_snd_devcap()`  
`SND_DEV_CAP`  
`SND_DEV_CM`  
`SND_SMAP`

---

**Note**

**CCITT G.711, G.721, G.723**: Sun and Microsoft have placed the source code of a portable implementation of these algorithms in the public domain. One place to ftp this source code from is `ftp.cwi.nl:/pub/audio/ccitt-adpcm.tar.Z`. 

---
**SYNTAX**

typedef struct _SND_DEV_CAP {
    char *hw_type; /* Hardware type */
    char *hw_subtype; /* Hardware subtype */
    SND_TRIGGER sup_triggers; /* Supported triggers */
    SND_LINE play_lines; /* Play Gain/mix lines */
    SND_LINE record_lines; /* Record Gain/mix lines */
    SND_GAIN_CMD sup_gain_cmds; /* Gain cmd mask */
    u_int16 num_gain_caps; /* Num gain caps */
    SND_GAIN_CAP *gain_caps; /* Ptr to gain cap array */
    u_int16 num_rates; /* Num sample rates */
    u_int32 *sample_rates; /* Ptr to sample rate array */
    u_int16 num_chan_info; /* Num channel info */
    u_int16 *channel_info; /* Ptr to channel info array */
    u_int16 num_cm; /* Num coding methods */
    SND_DEV_CM *cm_info; /* Ptr to coding method array */
} SND_DEV_CAP;

**DESCRIPTION**

This data structure defines the capabilities supported by a sound device. Appendix A: Sound Hardware Specifications gives detailed information about the capabilities of some specific MAUI sound drivers. Use _os_gs_snd_devcap() to retrieve this information from the driver.

- **hw_type** string indicates the class of hardware.
- **hw_subtype** string indicates the sub-class of hardware.
- **sup_triggers** indicates which triggers are supported by this device.
- **play_lines** indicates which mix lines control the gain level for _os_ss_snd_play().
record_lines indicates which mix lines control the gain level for _os_ss_snd_record().

sup_gain_cmds is a mask of the supported gain commands.

gain_caps is a pointer to an array of SND_GAIN_CAP structures, with num_gain_caps entries. This array has entries for each mixing line supported by the hardware.

sample_rates is a pointer to an array of the sample rates supported by the sound hardware, with num_rates entries.

channel_info is a pointer to an array of the various number of channels supported by the sound hardware, with num chan info entries. The channel_info array is typically one of the following:

- mono only: one element containing the value 1
- stereo only: one element containing the value 2
- stereo and mono: two elements, one containing the value 1 and the other containing the value 2

cm_info is a pointer to an array of SND_DEV_CM structures that describe coding methods supported by the sound hardware. The array has num_cm entries.

See Also
_os_gs_snd_devcap()
_os_ss_snd_cont()
_os_ss_snd_pause()
_os_ss_snd_play()
_os_ss_snd_record()
SND_DEV_CM
SND_GAIN_CAP
SND_LINE
SND_TRIGGER
SND_DEV_CM
Sound Device Coding Method

Syntax

typedef struct _SND_DEV_CM {
    SND_CM coding_method; /* Coding method */
    u_int32 sample_size; /* Number of bits per sample */
    u_int16 boundary_size; /* Boundary limitations */
} SND_DEV_CM;

Description

This data structure defines a coding method supported by a sound device. An array of sound coding methods supported by a driver is available via _os_gs_snd_devcap(). Appendix A: Sound Hardware Specifications gives detailed information about the capabilities of some specific MAUI drivers.

coding_method is the coding method of the sound data.
sample_size defines the size of each sample in bits.
boundary_size indicates the hardware memory boundary limitations in bytes. Memory submitted to the driver, via the buf field of a SND_SMAP, must start on the appropriate memory boundary and be a multiple of boundary_size. For example, if boundary_size is equal to 2, then buf should begin on an even byte boundary and buf_size should also be an even number of bytes. If boundary_size is equal to zero or 1, then there are no hardware limitations.

See Also

_os_gs_snd_devcap()
SND_CM
SND_DEV_CAP
SND_SMAP
SND_DEV_STATUS

Sound Device Status

Syntax

typedef struct _SND_DEV_STATUS {
    SND_STATUS status; /* Current status*/
    process_id play_pid; /* Current play PID*/
    process_id record_pid; /* Current record PID*/
    u_int16 num_gain; /* Num SND_GAIN structures */
    SND_GAIN *gain; /* Ptr to SND_GAIN array */
} SND_DEV_STATUS;

Description

This structure returns the current status of the sound device. This structure indicates what is being performed by which process. A pointer to this data structure is returned by _os_gs_snd_status().

status indicates the current status of the sound device.

If a play operation is active, the play_pid field contains the process id of the process that initiated the play. Otherwise, it contains zero.

If a record operation is active, the record_pid field contains the process id of the process that initiated the record. Otherwise it contains zero.

gain is a pointer to an array of SND_GAIN structures with num_gain entries. This array has entries for each mixing line that can be controlled on this hardware. Each entry indicates the current settings for each line.

See Also

_os_gs_snd_status()
SND_GAIN
SND_STATUS
**Syntax**

typedef struct _SND_GAIN {
    SND_LINE lines;  /* Mixer line mask */
    SND_GAIN_CMD cmd;  /* Type of gain */
    union {
        /* cmd == */
        SND_GAIN_UP up;  /* SND_GAIN_CMD_UP */
        SND_GAIN_DOWN down;  /* SND_GAIN_CMD_DOWN */
        SND_GAIN_MUTE mute;  /* SND_GAIN_CMD_MUTE */
        SND_GAIN_MONO mono;  /* SND_GAIN_CMD_MONO */
        SND_GAIN_STEREO stereo;  /* SND_GAIN_CMD_STEREO */
        SND_GAIN_XSTEREO xstereo;  /* SND_GAIN_CMD_XSTEREO */
    } param;
} SND_GAIN;

**Description**

This data structure is passed in _os_ss_snd_gain() to control input and output gain. The gain information specifies how much of each signal from the sound decoder/encoder contributes to the final channel outputs/inputs (such as speakers, headphones, microphones).

- **lines** specifies which mix lines are affected.
- **cmd** specifies how to modify the gain level.

The union **param** supplies the various parameters dependent on the value of **cmd**. The following table identifies which **cmd** values use which **params**: 
Table 4-3 Relationship Between cmd and param in SND_GAIN

<table>
<thead>
<tr>
<th>cmd</th>
<th>params</th>
</tr>
</thead>
<tbody>
<tr>
<td>SND_GAIN_CMD_UP</td>
<td>param.up.levels</td>
</tr>
<tr>
<td>SND_GAIN_CMD_DOWN</td>
<td>param.down.levels</td>
</tr>
<tr>
<td>SND_GAIN_CMD_RESET</td>
<td>No params</td>
</tr>
<tr>
<td>SND_GAIN_CMD_MUTE</td>
<td>para.mute.state</td>
</tr>
<tr>
<td>SND_GAIN_CMD_MONO</td>
<td>param.mono.m</td>
</tr>
<tr>
<td>SND_GAIN_CMD_STEREO</td>
<td>param.stereo.ll, param.stereo.rr</td>
</tr>
<tr>
<td>SND_GAIN_CMD_XSTEREO</td>
<td>param.xstereo.ll, param.xstereo.rr, param.xstereo.rl, param.xstereo.lr</td>
</tr>
</tbody>
</table>

See Also

_os_ss_snd_gain()
SND_GAIN_CMD
SND_GAIN_DOWN
SND_GAIN_MONO
SND_GAIN_MUTE
SND_GAIN_STEREO
SND_GAIN_UP
SND_GAIN_XSTEREO
SND_LINE
typedef struct _SND_GAIN_CAP {
    SND_LINE lines;/* Mask of mix lines */
    BOOLEAN sup_mute;/* Supports mute if TRUE */
    SND_GAIN_CMD default_type;
        /* Default gain type */
    u_int16 default_level;/* Default gain level */
    u_int16 zero_level;/* Gain setting where dB is */
         /* zero */
    u_int16 num_steps;/* Number of gain steps*/
    int16 step_size;/* Average size of each */
        /* step */
    int32 mindb;/* dB at SND_GAIN_MIN */
    int32 maxdb;/* dB at SND_GAIN_MAX */
} SND_GAIN_CAP;

Description

This data structure defines the gain capabilities of a set of mixer lines of a sound device. Appendix A: Sound Hardware Specifications gives detailed information about the capabilities of each specific MAUI sound driver. A pointer to an array of these data structures is returned as part of the SND_DEV_CAP structure returned in _os_gs_snd_devcap().

lines is a mask of one or more mix lines that share a SND_DEV_CAP description. A single SND_GAIN_CAP entry may describe more than one mixing line.

sup_mute is TRUE if the indicated mix lines support muting.

default_type specifies the default gain type for the indicated mix lines. This is the SND_GAIN_CMD that the mix lines revert to when SND_GAIN_CMD_RESET is specified in _os_ss_snd_gain().

default_level contains the driver's initial gain value for the indicated mix lines.
zero_level contains the level value where delta dB is zero (no gain or attenuation is applied to the line). Steps above this step are positive gain. Steps below this step are negative gain (attenuation).

num_steps is the number of actual gain or attenuation values supported by the hardware. Applications specify gain in levels between 0 and 127. The driver scales the requested level into the range of steps supported by the hardware.

step_size is the average size of each step in 100ths of a dB.

mindb is the delta dB in 100ths of a dB at level SND_GAIN_MIN.

maxdb is the delta dB in 100ths of a dB at level SND_GAIN_MAX.

The following two tables show example SND_GAIN_CAPs:

Table 4-4 Example A: SND_GAIN_CAP

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_PCM</td>
<td>Primary PCM CODEC.</td>
</tr>
<tr>
<td>sup_mute</td>
<td>TRUE</td>
<td>Mute is supported.</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_STEREO</td>
<td>Left and Right channels are supported.</td>
</tr>
<tr>
<td>default_level</td>
<td>SND_LEVEL_MAX</td>
<td>These lines start at and reset to level 127.</td>
</tr>
<tr>
<td>zero_level</td>
<td>SND_LEVEL_MIN</td>
<td>No gain applied when level is 0.</td>
</tr>
<tr>
<td>num_steps</td>
<td>64</td>
<td>The hardware supports 64 steps.</td>
</tr>
<tr>
<td>step_size</td>
<td>150</td>
<td>The average step size is 1.5 dB.</td>
</tr>
</tbody>
</table>
### Table 4-4 Example A: SND_GAIN_CAP (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mindb</td>
<td>9450</td>
<td>At level 0, an attenuation of 94.5 dB is applied.</td>
</tr>
<tr>
<td>maxdb</td>
<td>0</td>
<td>At level 127, no attenuation is applied.</td>
</tr>
</tbody>
</table>

The graph shows:
- 0 dB at max level 127
- -94.5 dB at min level 0
- 64 Steps

### Table 4-5 Example B: SND_GAIN_CAP

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_CD</td>
<td>SND_LINE_LINE</td>
</tr>
<tr>
<td>sup_mute</td>
<td>TRUE</td>
<td>Mute is supported.</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_STER EO</td>
<td>Left and Right channels are supported.</td>
</tr>
<tr>
<td>default_level</td>
<td>94</td>
<td>These lines start at and reset to level 94.</td>
</tr>
</tbody>
</table>
### Table 4-5 Example B: SND_GAIN_CAP (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero_level</td>
<td>94</td>
<td>No gain applied when level is 94.</td>
</tr>
<tr>
<td>num_steps</td>
<td>32</td>
<td>The hardware supports 32 steps.</td>
</tr>
<tr>
<td>step_size</td>
<td>150</td>
<td>The average step size is 1.5 dB.</td>
</tr>
<tr>
<td>mindb</td>
<td>-3450</td>
<td>At level 0, an attenuation of 34.5 dB is applied.</td>
</tr>
<tr>
<td>maxdb</td>
<td>120</td>
<td>At level 127, a gain of 12 dB is applied.</td>
</tr>
</tbody>
</table>

---

**See Also**

-os_gs_snd_devcap()

- BOOLEAN **MAUI Programming Reference Guide**
  - SND_DEV_CAP
  - SND_LINE
Syntax

typedef int {
    SND_GAIN_CMD_NONE, /* Gain command unknown */
    SND_GAIN_CMD_UP,   /* Increment gain n levels */
    SND_GAIN_CMD_DOWN, /* Decrement gain n levels */
    SND_GAIN_CMD_RESET, /* Reset to default level */
    SND_GAIN_CMD_MUTE, /* Set/Unset mute */
    SND_GAIN_CMD_MONO, /* Mono gain control */
    SND_GAIN_CMD_STEREO, /* Stereo gain control */
    SND_GAIN_CMD_XSTEREO/* Cross-stereo gain control */
} SND_GAIN_CMD;

Description

This integer defines the gain commands for modifying the gain level of
the mix lines. The values are powers of two to facilitate the specification
of their use in the sup_gain_cmds field of SND_GAIN_CAP.

SND_GAIN_CMD_NONE usually indicates that the gain command is not
set. A SND_GAIN->cmd of SND_GAIN_UP in _os_ss_snd_gain() indicates to increase the gain the number of levels specified in
SND_GAIN_UP.

A SND_GAIN->cmd of SND_GAIN_DOWN in _os_ss_snd_gain() indicates to decrease the gain the number of levels specified in
SND_GAIN_DOWN.

A SND_GAIN->cmd of SND_GAIN_CMD_RESET in
_os_ss_snd_gain() indicates to reset the gain level to the
default_level found in SND_GAIN_CAP.

A SND_GAIN->cmd of SND_GAIN_CMD_MUTE in
_os_ss_snd_gain() indicates to set or clear the mute bit according
to state specified in SND_GAIN_MUTE.
SND_GAIN_CMD_MONO specifies mono gain control. Parameters for SND_GAIN_CMD_MONO are specified by SND_GAIN_MONO.

SND_GAIN_STEREO specifies stereo gain control. Parameters for SND_GAIN_CMD_STEREO are specified by SND_GAIN_STEREO.

SND_GAIN_CMD_XTEREO is used for hardware that supports cross-gain or cross-attenuation. That is the ability to send the audio signal from the right channel to the left channel, and audio from the left channel to the right channel. Parameters for SND_GAIN_CMD_XSTEREO are specified by SND_GAIN_XSTEREO.

See Also
SND_GAIN
SND_GAIN_CAP
SND_GAIN_DOWN
SND_GAIN_MONO
SND_GAIN_MUTE
SND_GAIN_STEREO
SND_GAIN_UP
SND_GAIN_XSTEREO
Syntax
typedef struct _SND_GAIN_DOWN {
    u_int8 levels; /* Levels to decrement the */
    /* gain */
} SND_GAIN_DOWN;

Description
This data structure specifies how much to decrement the gain level when cmd of SND_GAIN is SND_GAIN_CMD_DOWN.

levels is the number of levels to decrement the gain level. This has the effect of decreasing the volume.

If decrementing the gain level by levels causes the gain level to drop below SND_LEVEL_MIN, the gain level is set to SND_LEVEL_MIN.

See Also
_os_ss_snd_gain()
SND_GAIN
SND_GAIN_CMD
SND_LEVEL_*
SND_GAIN_MONO
Mono Gain Parameters

Syntax

typedef struct _SND_GAIN_MONO {
    u_int8 m;    /* Gain of mono signal */
} SND_GAIN_MONO;

Description

This data structure specifies a specific mono gain level when cmd of SND_GAIN is SND_GAIN_CMD_MONO.

The gain level specifies how much of the mono signal from the sound decoder/encoder contributes to the final output or input (such as speaker, headphone, microphone).

The range of m is from SND_LEVEL_MIN to SND_LEVEL_MAX, inclusive. In general, higher m values are louder than lower m values. The significant values and translation of m to dB varies with the capabilities of the hardware. Use _os_gs_snd_devcap() and examine the SND_GAIN_CAP to determine the capabilities of a specific sound device.

Values in m with SND_LEVEL_MUTE bit set are considered mute. This allows the setting and clearing of the mute bit to enable and disable muting, without affecting the gain level of m.

See Also

_os_ss_snd_gain()  
SND_GAIN  
SND_GAIN_CAP  
SND_GAIN_CMD  
SND_LEVEL_*
**SND_GAIN_MUTE**

Gain Mute Parameters

**Syntax**

typedef struct _SND_GAIN_MUTE {
    BOOLEAN state; /*Mute state*/
} SND_GAIN_MUTE;

**Description**

This data structure specifies whether to mute or un-mute the line when cmd of SND_GAIN is SND_GAIN_CMD_MUTE.

If state is equal to FALSE, the line is un-muted. If state is equal to TRUE, the line is muted.

**See Also**

_os_ss_snd_gain()
SND_GAIN
SND_GAIN_CMD
SND_LEVEL_*

BOOLEAN  (See the MAUI Programming Reference Manual)
**SND_GAIN_STEREO**

Stereo Gain Parameters

### Syntax

typedef struct _SND_GAIN_STEREO {
    u_int8 ll;  /* Left input to left output */
    u_int8 rr;  /* Right input to right output */
} SND_GAIN_STEREO;

### Description

This data structure specifies a specific stereo gain level when cmd of SND_GAIN is SND_GAIN_CMD_STEREO.

The gain level specifies how much of each signal from the sound decoder/encoder contributes to the final left and right outputs or inputs (such as speaker, headphone, or microphone).

The range of each member (ll and rr) are from SND_LEVEL_MIN to SND_LEVEL_MAX inclusive. In general, higher values are louder than lower values. The significant values and translation to dB varies with the capabilities of the hardware. Use _os_gs_snd_devcap() to examine the SND_GAIN_CAP and determine the capabilities of a specific sound device.

Values in ll and rr with the SND_LEVEL_MUTE bit set are considered mute. This allows the setting and clearing of the mute bit to enable and disable muting without affecting the original values of ll and rr.

ll specifies the amount of audio to go from the left input to left output. rr specifies the amount of audio to go from the right input to right output. These fields allow control of the overall volume as well as adjusting balance.

### See Also

=os_ss_snd_gain()  
SND_GAIN  
SND_GAIN_CAP  
SND_GAIN_CMD  
SND_LEVEL_*
SND_GAIN_UP
Gain Up Parameters

Syntax

typedef struct _SND_GAIN_UP {
    u_int8 levels;/*Levels to increment the gain */
} SND_GAIN_UP;

Description

This data structure specifies how much to increment the gain level when cmd of SND_GAIN is SND_GAIN_CMD_UP.

levels is the number of levels to increment the gain level. This has the effect of increasing the volume. If incrementing the gain level by levels causes the gain level rise above SND_LEVEL_MAX, the gain level is set to SND_LEVEL_MAX.

See Also

_os_ss_snd_gain()
SND_GAIN
SND_GAIN_CMD
SND_LEVEL_*
SND_GAIN_XSTEREO
Cross-Stereo Gain Parameters

Syntax

typedef struct _SND_GAIN_STEREO {
    u_int8 ll; /* Left input to left output */
    u_int8 rr; /* Right input to right output */
    u_int8 rl; /* Right input to left output */
    u_int8 lr; /* Left input to right output */
} SND_GAIN_STEREO;

Description

This data structure specifies a specific stereo gain level when cmd of SND_GAIN is equal to SND_GAIN_CMD_XSTEREO.

The gain level specifies how much of each signal from the sound decoder/encoder contributes to the final left and right outputs and inputs (such as speaker, headphone, and microphone).

The range of each member (ll, rr, rl, and lr) is from SND_LEVEL_MIN to SND_LEVEL_MAX inclusive. In general, higher values are louder than lower values. The significant values and translation to dB varies with the capabilities of the hardware. Use _os_gw_snd_dev_cap() and examine the SND_GAIN_CAP to determine the capabilities of a specific sound device.

Values in ll, rr, rl, and lr with the SND_LEVEL_MUTE bit set are considered mute. This allows the setting and clearing of the mute bit to enable and disable muting, without affecting the original values of ll, rr, rl, and lr.

ll specifies the amount of audio going from the left input to left output.

rr specifies the amount of audio going from the right input to the right output.

rl specifies the amount of audio going from the right input to left output.
lr specifies the amount of audio going from the left input to right output.

These fields allow control of the overall volume as well as adjusting balance.

See Also
_os_ss_snd_gain()
SND_GAIN
SND_GAIN_CAP
SND_GAIN_CMD
SND_LEVEL_*
**SND_LEVEL_***

Gain Level Constants

**Syntax**

SND_LEVEL_*

**Description**

SND_LEVEL_ is a prefix used to define a set of constant values that may be used to specify gain levels as shown in the following table. These constants are commonly used in SND_GAIN_DOWN, SND_GAIN_MONO, SND_GAIN_MUTE, SND_GAIN_STEREO, SND_GAIN_UP, and SND_GAIN_XSTEREO.

**Table 4-6 SND_LEVEL_* Definitions**

<table>
<thead>
<tr>
<th>Define Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SND_LEVEL_MIN</td>
<td>0</td>
<td>Defines the minimum (quietest) gain level.</td>
</tr>
<tr>
<td>SND_LEVEL_MAX</td>
<td>127</td>
<td>Defines the maximum (loudest) gain level.</td>
</tr>
<tr>
<td>SND_LEVEL_MUTE</td>
<td>0x80</td>
<td>Mask value used to mute a line.</td>
</tr>
</tbody>
</table>

SND_LEVEL_MUTE may be logically or’ed (|) with a gain level to mute the line without affecting the current gain level. For example:

```c
SN_GAIN in_gain, out_gain;
in_gain.param.mono.m |= SND_LEVEL_MUTE;
out_gain.param.stereo.ll |= SND_LEVEL_MUTE;
out_gain.param.stereo.rr |= SND_LEVEL_MUTE;
```

The negation (~) of this mask may be logically and’ed (&) with gain to un-mute the line without affecting the current gain level. For example:
SND_GAIN in_gain, out_gain;
in_gain.param.mono.m &= ~SND_LEVEL_MUTE;
out_gain.param.stereo.ll &= ~SND_LEVEL_MUTE;
out_gain.param.stereo.rr &= ~SND_LEVEL_MUTE;

See Also
_os_ss_snd_gain()
SND_GAIN
SND_GAIN_DOWN
SND_GAIN_MONO
SND_GAIN_MUTE
SND_GAIN_STEREO
SND_GAIN_UP
SND_GAIN_XSTEREO
**SND_LINE**
Gain/Mixer Line Types

**Syntax**

typedef int {
    SND_LINE_VOLUME,  /* Master output */
    SND_LINE_BASS,    /* Bass */
    SND_LINE_TREBLE,  /* Treble */
    SND_LINE_SYNTH,   /* Synthesizer input */
    SND_LINE_PCM,     /* PCM/CODEC */
    SND_LINE_SPEAKER, /* PC speaker */
    SND_LINE_LINE,    /* Line input */
    SND_LINE_MIC,     /* Microphone input */
    SND_LINE_CD,      /* CD input */
    SND_LINE_IMIX,    /* Recording monitor */
    SND_LINE_OMIX,    /* Loopback */
    SND_LINE_ALTPCM,  /* Alternative PCM/CODEC */
    SND_LINE_RECLEV,  /* Encoder level */
    SND_LINE_IGAIN,   /* Input gain */
    SND_LINE_OGAIN,   /* Output gain */
    SND_LINE_LINE1,   /* Lines 1-3 are generic mixer lines */
    SND_LINE_LINE2,   
    SND_LINE_LINE3,   
    SND_NUM_LINES     /* Number of mixer lines */
    SND_MAX_LINES     /* Maximum mixer lines */
    SND_LINE_MASK_ALL /* Mask to select all mixer lines */
} SND_LINE;

**Description**

This integer defines the valid mixer line types. The values are powers of two. Therefore, you may combine them safely. It is used in
SND_DEV_CAP, SND_GAIN, and SND_GAIN_CAP. These definitions are similar to LINUX's SOUND_MIXER_* definitions.
SND_LINE_VOLUME is the master output level (headphone or line out volume).

SND_LINE_BASS controls the bass level of all the output lines.

SND_LINE_TREBLE controls the treble level of all the output lines.

SND_LINE_SYNTH controls the synthesizer input (FM, wave table) of the sound card.

SND_LINE_PCM is the output level for the audio (CODEC, PCM, ADC) line.

SND_LINE_SPEAKER is the output level of the PC speaker signals. This is typically mono.

SND_LINE_LINE is the input level for the line-in jack.

SND_LINE_MIC is the input level for the signal coming from the microphone-in jack.

SND_LINE_CD is the level for signal connected to the CD audio input.

SND_LINE_IMIX is the recording monitor. For example, on PAS16 and some other cards, this controls the output gain (headphone jack) of the selected recording sources while recording. This line only has effect when recording.

SND_LINE_OMIX controls the loopback of output to input.

SND_LINE_ALTPCM controls the alternative CODEC line such as the SB emulation of the PAS16 board.

SND_LINE_RECLEV is the global record level setting. On the SB16 card, this controls the input gain, which has 4 possible levels.

SND_LINE_IGAIN is the input gain control.

SND_LINE_OGAIN is the output gain control.

SND_LINE_LINE1, SND_LINE_LINE2, and SND_LINE_LINE3 are generic mixer lines that are used in cases when precise meaning of a physical mixer line is not known. Actual meaning of these signals are vendor-defined. Usually these lines are connected to synth, line-in, and CD inputs of the card, but the order of the assignment is not known to the driver.

SND_LINE_MASK_ALL is a mask to select all mixer lines.
See Also
SND_DEV_CAP
SND_GAIN
SND_GAIN_CAP
**Syntax**

```c
typedef struct _SND_SMAP {
    SND_TRIGGER trig_status;
    /* Current sound map status */
    SND_TRIGGER trig_mask; /* Signal trigger mask */
    signal_code trig_signal;
    /* Signal to send on */
    /* triggers */
    error_code err_code; /* Error code on termination */
    SND_CM coding_method; /* Coding method */
    u_int8 num_channels; /* Number of channels */
    u_int32 sample_size; /* Number of bits per sample */
    u_int32 sample_rate; /* Number of samples per sec */
    u_char *buf; /* Sound data buffer */
    u_int32 buf_size; /* Size of sound data buffer */
    u_int32 cur_offset; /* Current offset into “buf” */
    u_int32 loop_start; /* Offset to start of loop */
    u_int32 loop_end; /* Offset to end of loop */
    u_int32 loop_count; /* Num of times to play loop */
    u_int32 loop_counter; /* Num times loop has played */
    SND_SMAP *next; /* Link to next SND_SMAP */
} SND_SMAP;
```

**Description**

This data structure defines the sound map. This object is created by the application and is used to control play and record operations.

- **trig_status** indicates the current sound map status. It is modified as the status changes. See **SND_TRIGGER** for a description of each value.

- The specified **trig_signal** is sent when the sound operation (_os_ss_snd_play() or _os_ss_snd_record()) completes an activity that satisfies a trigger specified by **trig_mask**. If the specified **trig_mask** is equal to **SND_TRIG_NONE** or **trig_signal** is equal to zero, then no signals are sent.
If an error occurs, \texttt{err_code} is set to an appropriate error code. If an operation is aborted, \texttt{err_code} is set to \texttt{EOS MAUI_ABORT}. Otherwise, \texttt{err_code} is set to \texttt{SUCCESS} at the end of the sound operation.

\texttt{coding_method} defines the audio encoding format for the sound map. \texttt{num_channels} is the number of channels. Mono data requires one channel. Stereo data requires two channels. Currently all coding methods are defined such that the left stereo channel is always ordered before the right stereo channel.

There are two main parameters that effect the quality of the sound. First, is the \texttt{sample_rate} which is the number of samples per second. Typical values include \texttt{4000, 8000, 11025, 22050, 44100 and 48000}. The second parameter that affects the quality is \texttt{sample_size}. \texttt{sample_size} is the number of bits per sample. Typical values are 8 or 16 bits per sample. The number of bits affect the dynamic range of the sample and the signal to noise ratio.

\texttt{buf} is a pointer to a buffer containing data for a play operation or sufficient space for a record operation. The memory pointed to by \texttt{buf} should start on a multiple of \texttt{SND_DEV_CM->boundary_size}. The size of the buffer, in bytes, is set in \texttt{buf_size} and should also be a multiple of \texttt{SND_DEV_CM->boundary_size}.

\texttt{cur_offset} specifies the offset in bytes within \texttt{buf} for the current sound operation. It is updated throughout the I/O operation by the sound driver.

\texttt{loop_start}, \texttt{loop_end}, \texttt{loop_count} and \texttt{loop_counter} are used to specify the area of the buffer to play and how many times to play it. They are not used in record operations and must be set to zero when submitted to \texttt{os_ss_snd_record()}. \texttt{loop_start} specifies the start of the loop as an offset in bytes within \texttt{buf}. \texttt{loop_end} specifies the end of the loop as an offset in bytes within \texttt{buf}. \texttt{loop_count} indicates the desired number of times to execute the loop.
loop_counter indicates the current number of times the loop has been executed. This field is set to zero by the sound driver when the sound map is accepted by the sound driver.

next may point to another sound map. In this case, the sound operation continues with the next SND_SMAP when the first one is finished. The trig_mask and trig_signal are consulted regardless of whether next is set or not. If this SND_SMAP is not linked to another SND_SMAP, next must be NULL.

Linked sound maps may have different encoding parameters (coding_method, num_channels, sample_size, sample_rate), although most hardware requires some calibration time to switch to the new formats, which may result in a delay.

See Also
_os_ss_snd_play()
_os_ss_snd_record()
SND_CM
SND_DEV_CM
SND_TRIGGER
Data Type Reference

SND_STATUS

Device Status

Syntax

typedef int {
    SND_STATUS_IDLE, /* Device is idle */
    SND_STATUS_PLAY, /* Play operation is active */
    SND_STATUS_PLAY_PAUSED,/* Play is paused */
    SND_STATUS_RECORD, /* Record operation is active */
    SND_STATUS_RECORD_PAUSED,/* Record is paused */
    SND_STATUS_BUSY /* The buffer is busy */
} SND_STATUS;

Description

This integer defines the valid values for the status field in
SND_DEV_STATUS. SND_DEV_STATUS is returned from
_os_ss_snd_status() and indicates the current state of the sound
hardware. The values are powers of two and may be combined safely.

SND_STATUS_IDLE (no status bits are set) indicates the sound device
is not busy.

The SND_STATUS_PLAY bit is set when a play operation is active. This
bit is set during a play operation from the SND_TRIG_START event to
the SND_TRIG_FINISH event.

The SND_STATUS_PLAY_PAUSED bit is set when a play operation is
paused. This bit is never set if the SND_STATUS_PLAY bit is not set.

The SND_STATUS_RECORD bit is set when a record operation is active.
This bit is set during a record operation from the SND_TRIG_START
event to the SND_TRIG_FINISH event.

The SND_STATUS_RECORD_PAUSED bit is set when a record operation
is paused. This bit is never set if the SND_STATUS_RECORD bit is not
set.

The SND_STATUS_BUSY bit is set when the driver is processing a
sound operation.
For illustrations of `SND_TRIGGER` in sound operations see the figure "Play Operation Status and Trigger Points." on page 29 and "Record Operation Status and Trigger Points" on page 31, respectively.

See Also

`_os_ss_snd_play()`  
`_os_ss_snd_record()`  
SND_SMAP
**Syntax**

typedef int {
    SND_TRIG_NONE,     /* Mask for no triggers */
    SND_TRIG_START,    /* Audible start */
    SND_TRIG_FINISH,   /* Audible finish */
    SND_TRIG_BUSY,     /* Buffer Busy */
    SND_TRIG_READY,    /* Buffer ready */
    SND_TRIG_ANY       /* Mask for all triggers */
} SND_TRIGGER;

**Description**

This integer defines the trigger events in a sound operation (`_os_ss_snd_play()` or `_os_ss_snd_record()`) capable of generating signals. The values are powers of two and may be combined safely.

SND_TRIG_NONE is a mask value indicating interest in no triggers.

If the SND_TRIG_START bit is set for a play operation, this indicates that the output device is actively playing (it can be heard). For a record operation, this bit indicates that the driver has started accepting sound input.

If the SND_TRIG_FINISH bit is set for a play operation, this indicates that the driver has completed the play and no more sound is being produced. For a record operation, this bit indicates that the driver has stopped accepting sound input.

If the SND_TRIG_BUSY bit is set for a play operation, this indicates that the driver has started consuming the data in the buffer. For a record operation, this bit indicates that the driver has started filling the buffer.

If the SND_TRIG_READY bit is set for a play operation, this indicates that the driver has finished consuming the data in the buffer and is ready to accept the next buffer. For a record operation, this bit indicates that the driver has finished filling the buffer and is ready to use the next buffer.
SND_TRIG_ANY is a mask value indicating interest in all triggers. For illustrations of SND_TRIGGER in sound operations see the figure "Play Operation Status and Trigger Points." on page 29 and "Record Operation Status and Trigger Points" on page 31.

See Also
_os_ss_snd_play()
_os_ss_snd_record()
SND_SMAP
Appendix A: Sound Hardware Specifications

This appendix contains the hardware specifications for the following sound device:

Crystal Semiconductor CS4231A
Overview

This document describes the hardware specifications for the Crystal Semiconductor CS4231A driver (named sd_cs). The hardware sub-type defines the board configuration. This specification should be used in conjunction with the MAUI Sound Driver Interface.
Device Capabilities

Information about the hardware capabilities is determined by calling `_os_gs_snd_devcap()`. This function returns a data structure formatted as shown in the following table.

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hw_type</td>
<td>“CS4231”</td>
<td>Hardware type</td>
</tr>
<tr>
<td>hw_subtype</td>
<td>“CS4231A”</td>
<td>Hardware sub-type</td>
</tr>
<tr>
<td>sup_triggers</td>
<td>SND_TRIG_ANY</td>
<td>Supported triggers</td>
</tr>
<tr>
<td>play_lines</td>
<td>SND_LINE_SPEAKER</td>
<td>Play gain/mix lines</td>
</tr>
<tr>
<td></td>
<td>SND_LINE_VOLUME</td>
<td></td>
</tr>
<tr>
<td>record_lines</td>
<td>SND_LINE_MIC</td>
<td>Record gain/mix lines</td>
</tr>
<tr>
<td></td>
<td>SND_LINE_LINE3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SND_LINE_LINE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SND_LINE_PCM</td>
<td></td>
</tr>
<tr>
<td>sup_gain_cmds</td>
<td>SND_GAIN_CMD_UP</td>
<td>Mask of supported gain commands</td>
</tr>
<tr>
<td></td>
<td>SND_GAIN_CMD_DOWN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SND_GAIN_CMD_RESET</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SND_GAIN_CMD_MONO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SND_GAIN_CMD_STEREO</td>
<td></td>
</tr>
<tr>
<td>num_gain_caps</td>
<td>7</td>
<td>Number of SND_GAIN_CAPS</td>
</tr>
</tbody>
</table>
### Table A-1 Data Returned in SND_DEV_CAP (continued)

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gain_caps</code></td>
<td>See paragraph Gain</td>
<td>Pointer to SAND_GAIN_CAP array</td>
</tr>
<tr>
<td></td>
<td>Capabilities Array</td>
<td></td>
</tr>
<tr>
<td><code>num_rates</code></td>
<td>14</td>
<td>Number of sample rates</td>
</tr>
<tr>
<td><code>sample_rates</code></td>
<td>See paragraph Sample</td>
<td>Pointer to sample rate array</td>
</tr>
<tr>
<td></td>
<td>Rates</td>
<td></td>
</tr>
<tr>
<td><code>num_chan_info</code></td>
<td>2</td>
<td>Number of channel info entries</td>
</tr>
<tr>
<td><code>channel_info</code></td>
<td>See paragraph Number</td>
<td>Pointer to an array of supported</td>
</tr>
<tr>
<td></td>
<td>of Channels</td>
<td>supported num_channels</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>num_cm</code></td>
<td>6</td>
<td>Number of coding methods</td>
</tr>
<tr>
<td><code>cm_info</code></td>
<td>See paragraph Encoding</td>
<td>Pointer to coding method array</td>
</tr>
<tr>
<td></td>
<td>and Decoding Formats</td>
<td></td>
</tr>
</tbody>
</table>

---

**For More Information**

See `SND_DEV_CAP` in chapter 4 for more information regarding this data structure.
Gain Capabilities Array

The preceding table shows the various gain capabilities for the Crystal Semiconductor CS4231A. This information is pointed to by the gain_cap member of the SND_DEV_CAP data structure. The following seven tables (Tables 2 through 8) describe the gain capabilities.

Table A-2  L/R DAC Attenuator (I6) L/R DAC Attenuator (I7)

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_PCM</td>
</tr>
<tr>
<td>sup_mute</td>
<td>TRUE</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_STEREO</td>
</tr>
<tr>
<td>default_level</td>
<td>SND_LEVEL_MAX</td>
</tr>
<tr>
<td>zero_level</td>
<td>SND_LEVEL_MIN</td>
</tr>
<tr>
<td>num_steps</td>
<td>64</td>
</tr>
<tr>
<td>step_size</td>
<td>150</td>
</tr>
<tr>
<td>mindb</td>
<td>-9450</td>
</tr>
<tr>
<td>maxdb</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>63</td>
<td>-94.5 dB</td>
<td>default_level</td>
</tr>
<tr>
<td>2-3</td>
<td>62</td>
<td>-93.0 dB</td>
<td></td>
</tr>
<tr>
<td>4-5</td>
<td>61</td>
<td>-91.5 dB</td>
<td></td>
</tr>
<tr>
<td>6-7</td>
<td>60</td>
<td>-90.0 dB</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
### Sound Hardware Specifications

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>120-121</td>
<td>3</td>
<td>-4.5 dB</td>
<td></td>
</tr>
<tr>
<td>122-123</td>
<td>2</td>
<td>-3.0 dB</td>
<td></td>
</tr>
<tr>
<td>124-125</td>
<td>1</td>
<td>-1.5 dB</td>
<td></td>
</tr>
<tr>
<td>126-127</td>
<td>0</td>
<td>0.0 dB</td>
<td>zero_level</td>
</tr>
</tbody>
</table>
### Table A-3  L/R Auxiliary #1 (I2, I3), L/R Auxiliary #2 (I4, I5), L/R Line Mix Gain (I18, I19)

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_CD</td>
</tr>
<tr>
<td>sup_mute</td>
<td>TRUE</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_STEREO</td>
</tr>
<tr>
<td>default_level</td>
<td>94</td>
</tr>
<tr>
<td>zero_level</td>
<td>94</td>
</tr>
<tr>
<td>num_steps</td>
<td>32</td>
</tr>
<tr>
<td>step_size</td>
<td>150</td>
</tr>
<tr>
<td>mindb</td>
<td>-3450</td>
</tr>
<tr>
<td>maxdb</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>31</td>
<td>-34.5 dB</td>
<td></td>
</tr>
<tr>
<td>4-6</td>
<td>30</td>
<td>-33.0 dB</td>
<td></td>
</tr>
<tr>
<td>7-10</td>
<td>29</td>
<td>-31.5 dB</td>
<td></td>
</tr>
<tr>
<td>11-14</td>
<td>28</td>
<td>-30.5 dB</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>89-92</td>
<td>9</td>
<td>-1.5 dB</td>
<td></td>
</tr>
<tr>
<td>93-96</td>
<td>8</td>
<td>0 dB</td>
<td>zero_level,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_level</td>
</tr>
</tbody>
</table>
### Sound Hardware Specifications

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>97-100</td>
<td>7</td>
<td>1.5 dB</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>113-116</td>
<td>3</td>
<td>7.5 dB</td>
<td></td>
</tr>
<tr>
<td>117-120</td>
<td>2</td>
<td>9.0 dB</td>
<td></td>
</tr>
<tr>
<td>121-124</td>
<td>1</td>
<td>10.5 dB</td>
<td></td>
</tr>
<tr>
<td>125-127</td>
<td>0</td>
<td>12.0 dB</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-4  L/R ADC Gain (I0, I1)

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_IGAIN</td>
</tr>
<tr>
<td>sup_mute</td>
<td>FALSE</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_STEREO</td>
</tr>
<tr>
<td>default_level</td>
<td>SND_LEVEL_MIN</td>
</tr>
<tr>
<td>zero_level</td>
<td>SND_LEVEL_MIN</td>
</tr>
<tr>
<td>num_steps</td>
<td>16</td>
</tr>
<tr>
<td>step_size</td>
<td>150</td>
</tr>
<tr>
<td>mindb</td>
<td>0</td>
</tr>
<tr>
<td>maxdb</td>
<td>2250</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td>0</td>
<td>0 dB</td>
<td>zero_level,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_level</td>
</tr>
<tr>
<td>5-12</td>
<td>1</td>
<td>1.5 dB</td>
<td></td>
</tr>
<tr>
<td>13-21</td>
<td>2</td>
<td>3.0 dB</td>
<td></td>
</tr>
<tr>
<td>22-30</td>
<td>3</td>
<td>4.5 dB</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Sound Hardware Specifications

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>98-105</td>
<td>12</td>
<td>18.0 dB</td>
<td></td>
</tr>
<tr>
<td>106-114</td>
<td>13</td>
<td>19.5 dB</td>
<td></td>
</tr>
<tr>
<td>115-122</td>
<td>14</td>
<td>21.0 dB</td>
<td></td>
</tr>
<tr>
<td>124-127</td>
<td>15</td>
<td>22.5 dB</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-5  L/R Mic Gain Enable (I0, I1)

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_MIC</td>
</tr>
<tr>
<td>sup_mute</td>
<td>FALSE</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_STEREO</td>
</tr>
<tr>
<td>default_level</td>
<td>SND_LEVEL_MAX</td>
</tr>
<tr>
<td>zero_level</td>
<td>SND_LEVEL_MIN</td>
</tr>
<tr>
<td>num_steps</td>
<td>2</td>
</tr>
<tr>
<td>step_size</td>
<td>200</td>
</tr>
<tr>
<td>mindb</td>
<td>0</td>
</tr>
<tr>
<td>maxdb</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-63</td>
<td>0</td>
<td>0</td>
<td>zero_level</td>
</tr>
<tr>
<td>64-127</td>
<td>20</td>
<td>1</td>
<td>default_level</td>
</tr>
</tbody>
</table>
**Table A-6  Mono In/Out Speaker (I26)**

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_SPEAKER</td>
</tr>
<tr>
<td>sup_mute</td>
<td>TRUE</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_MONO</td>
</tr>
<tr>
<td>default_level</td>
<td>SND_LEVEL_MAX</td>
</tr>
<tr>
<td>zero_level</td>
<td>SND_LEVEL_MIN</td>
</tr>
<tr>
<td>num_steps</td>
<td>16</td>
</tr>
<tr>
<td>step_size</td>
<td>300</td>
</tr>
<tr>
<td>mindb</td>
<td>-4500</td>
</tr>
<tr>
<td>maxdb</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td>0</td>
<td>0 dB</td>
<td>zero_level,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default_level</td>
</tr>
<tr>
<td>5-12</td>
<td>1</td>
<td>-3.0 dB</td>
<td></td>
</tr>
<tr>
<td>13-21</td>
<td>2</td>
<td>-6.0 dB</td>
<td></td>
</tr>
<tr>
<td>22-30</td>
<td>3</td>
<td>-9.0 dB</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
## Sound Hardware Specifications

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>98-105</td>
<td>12</td>
<td>-36.0 dB</td>
<td></td>
</tr>
<tr>
<td>106-114</td>
<td>13</td>
<td>-39.0 dB</td>
<td></td>
</tr>
<tr>
<td>115-122</td>
<td>14</td>
<td>-42.0 dB</td>
<td></td>
</tr>
<tr>
<td>124-127</td>
<td>15</td>
<td>-45.0 dB</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-7  Loopback Attenuation (I13)

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_IMIX</td>
</tr>
<tr>
<td>sup_mute</td>
<td>TRUE</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_MONO</td>
</tr>
<tr>
<td>default_level</td>
<td>SND_LEVEL_MAX</td>
</tr>
<tr>
<td>zero_level</td>
<td>SND_LEVEL_MIN</td>
</tr>
<tr>
<td>num_steps</td>
<td>64</td>
</tr>
<tr>
<td>step_size</td>
<td>150</td>
</tr>
<tr>
<td>mindb</td>
<td>-9450</td>
</tr>
<tr>
<td>maxdb</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>HW</th>
<th>Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>63</td>
<td>-94.5 dB</td>
<td>default_level</td>
</tr>
<tr>
<td>2-3</td>
<td>62</td>
<td>-93.0 dB</td>
<td></td>
</tr>
<tr>
<td>4-5</td>
<td>61</td>
<td>-91.5 dB</td>
<td></td>
</tr>
<tr>
<td>6-7</td>
<td>60</td>
<td>-90.0 dB</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>120-121</td>
<td>3</td>
<td>-4.5 dB</td>
<td></td>
</tr>
<tr>
<td>122-123</td>
<td>2</td>
<td>-3.0 dB</td>
<td></td>
</tr>
<tr>
<td>124-125</td>
<td>1</td>
<td>-1.5 dB</td>
<td></td>
</tr>
<tr>
<td>126-127</td>
<td>0</td>
<td>0.0 dB</td>
<td>zero_level</td>
</tr>
</tbody>
</table>
### Table A-8  On some boards the I10 register can mute the line out

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>SND_LINE_VOLUME</td>
</tr>
<tr>
<td>sup_mute</td>
<td>TRUE</td>
</tr>
<tr>
<td>default_type</td>
<td>SND_GAIN_CMD_MUTE</td>
</tr>
<tr>
<td>default_level</td>
<td>0</td>
</tr>
<tr>
<td>zero_level</td>
<td>0</td>
</tr>
<tr>
<td>num_steps</td>
<td>0</td>
</tr>
<tr>
<td>step_size</td>
<td>0</td>
</tr>
<tr>
<td>mindb</td>
<td>0</td>
</tr>
<tr>
<td>maxdb</td>
<td>0</td>
</tr>
</tbody>
</table>

### Note

See `SND_GAIN_CAP` in Chapter 4: Data Type Reference for more information about this data structure.
Sample Rates

The following table shows the supported sample rates for the Crystal Semiconductor CS4231A. This information is pointed to by the sample_rates member of the SND_DEV_CAP data structure.

**Table A-9  Supported Sample Rates**

<table>
<thead>
<tr>
<th>Sample Rate</th>
<th>Sample Rate</th>
<th>Sample Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>5510 Hz</td>
<td>11025 Hz</td>
<td>32000 Hz</td>
</tr>
<tr>
<td>6620 Hz</td>
<td>16000 Hz</td>
<td>33075 Hz</td>
</tr>
<tr>
<td>8000 Hz</td>
<td>18900 Hz</td>
<td>37800 Hz</td>
</tr>
<tr>
<td>9600 Hz</td>
<td>22050 Hz</td>
<td>44100 Hz</td>
</tr>
<tr>
<td></td>
<td>27420 Hz</td>
<td>48000 Hz</td>
</tr>
</tbody>
</table>

Number of Channels

Table 10 shows the different supported number of channels for the CS4231A. This information is pointed to by the channel_info member of the SND_DEV_CAP data structure.

**Table A-10  Supported Number of Channels**

<table>
<thead>
<tr>
<th>Channels</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Stereo (default)</td>
</tr>
<tr>
<td>1</td>
<td>Mono</td>
</tr>
</tbody>
</table>
Encoding and Decoding Formats

Table 11 shows the supported encoding and decoding formats for the Crystal Semiconductor CS4231A. The first entry in the table is the default format. This information is pointed to by the cm_info member of the SND_DEV_CAP data structure.

Table A-11 Supported Encoding and Decoding Formats

<table>
<thead>
<tr>
<th>Coding Method</th>
<th>Sample Size</th>
<th>Bndry Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SND_CM_PCM_ULAW</td>
<td>8</td>
<td>2</td>
<td>8-bit µLaw companded</td>
</tr>
<tr>
<td>SND_CM_PCM_ALAW</td>
<td>8</td>
<td>2</td>
<td>8-bit ALaw companded</td>
</tr>
<tr>
<td>SND_CM_PCM_ULINEAR</td>
<td>8</td>
<td>2</td>
<td>8-bit Linear unsigned</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>little-endian</td>
</tr>
<tr>
<td>SND_CM_PCM_SLINEAR</td>
<td>16</td>
<td>4</td>
<td>16-bit Linear (two’s complement)</td>
</tr>
</tbody>
</table>
A
Sound Hardware Specifications

Using the Sound Driver Interface
Index

Symbols

_os_close() 42
_os_detach() 43
_os_gs_devcap() 16
_os_gs_snd_compat() 43
_os_gs_snd_devcap() 45
_os_gs_snd_status() 24, 47
_os_open() 22, 51
_os_ss_relea() 51
_os_ss_sendsig() 25, 52
_os_ss_snd_abort() 36
_os_ss_snd_cont() 35, 56
_os_ss_snd_gain() 31, 56
_os_ss_snd_pause() 35, 60
_os_ss_snd_play() 27, 62
_os_ss_snd_record() 29, 66

A

Abort Active Sound Operation 54
ADPCM 19
A-Law 19

B

BLOCK_FINISH 28, 30
BLOCK_START 28, 30
Blocking Types 74
Buffers 11, 14
    Current offset 14
    ID 14
    size 14
### Capabilities
- **Device** 16
- **Sound Device** 45

#### CD4231A
- **Gain Capabilities** 123

- **Check**
  - **compatibility** 16
  - **status** 16

- **Check device capabilities** 16

- **Close** 37

- **Close Path to Sound Device** 42

- **Coding method** 13

- **Coding Methods**
  - **Supported formats** 19

- **Compatibility** 16

- **Compatibility Level** 43

- **Continue Active Sound Operation** 56

- **Cross-Stereo Gain Parameters** 105

- **Crystal Semiconductor CS4231A** 120

- **CS4231A** 120

  - **Device Capabilities** 121
  - **Encoding and Decoding Formats** 135
  - **Sample Rates** 134

### Data Type Reference
- **71**

- **Device Capabilities** 16

- **Device capabilities** 16

- **Device Status** 115

- **Driver**
  - **Compatibility** 43

- **Driver Compatibility Level** 20

### Encoding Parameters
- **13**

  - **Coding method** 13
Number of Channel 13
Sample rate 13
Sample size 13
Encoding parameters 11
Entry Points
  Primary 9
  Setstat 9
Error signal 12

Functions 9

Gain Down Parameters 100
Gain Up Parameters 104
Get Driver Compatibility Level 43
Get Sound Device Capabilities 45
Get Sound Device Status 47

Loop count 15
Loop counter 15
Loop end 15
Loop start 15
Looping 11, 15

m-Law 19
Mono Gain Parameters 101

Next 11, 15
NOBLOCK 28, 30
Number of Channels 13
Open 22
Open Path to Sound Device 49
os_ss_snd_abort() 54

Pause Active Sound Operation 60
PCM Linear 19
Play
   Abort 36
   Pause 35
Play command 27
Play PID 24
Play Sound to Sound Decoder 62
Playing a Sound File 27
Preparing the Sound Device for Use 22

Record
   Gain Control 31
Record command 29
Record PID 24
Record Sound 66
Release 37
Release Device 51

Sample rate 13
Sample size 13
Send Signal 24
Send Signal on Device Ready 52
Set Gain 58
SND_BLOCK_TYPE 74
SND_CM 76
SND_CM_ADPCM_G721 86
SND_CM_ADPCM_G723 87
SND_CM_ADPCM_IMA 85
SND_CM_PCM_LINEAR 80
SND_CM_PCM_ULAW 84
SND_CMUNKNOWN 80
SND_DEV_CAP 16
SND_DEV_CM 90
SND_DEV_STATUS 91
SND_GAIN 92
SND_GAIN_DOWN 100
SND_GAIN_MIN 101
SND_GAIN_MONO 101
SND_GAIN_MUTE 103
SND_GAIN_STEREO 103
SND_GAIN_TYPE 104
SND_GAIN_UP 104
SND_GAIN_XSTEREO 105
SND_SMAP 112
SND_STATUS 115
SND_TRIGGER 117
Software compatibility 16
Sound Device
   Capabilities 16, 45
   Close 37, 42
   Play PID 24
   Preparing for Use 22
   Record PID 24
   Release 37, 51
   Status 16, 47
Sound Device Coding Methods 90
Sound driver
   Compatibility 16
   compatibility level 20
Sound Driver Interface
   Overview 8
Sound Hardware Specifications 119
Sound Map 112
Sound Maps 10
Specification, CS4231A 120
Status
   Sound Device 47
   Sound device 16
Stereo Gain Parameters  103

Triggers  11,  12,  117
  error signal  12
  mask  12
  signal  12
  status  12