NAME

audio – generic audio device interface

OVERVIEW

The audio interface described below is an uncommitted interface and may be replaced in the future.

An audio device is used to play and/or record a stream of audio data. Since a specific audio device may not support all of the functionality described below, refer to the device-specific manual pages for a complete description of each hardware device. An application can use the **AUDIO_GETDEV** **ioctl**(2) to determine the current audio hardware associated with **/dev/audio**.

AUDIO FORMATS

Digital audio data represents a quantized approximation of an analog audio signal waveform. In the simplest case, these quantized numbers represent the amplitude of the input waveform at particular sampling intervals. In order to achieve the best approximation of an input signal, the highest possible sampling frequency and precision should be used. However, increased accuracy comes at a cost of increased data storage requirements. For instance, one minute of monaural audio recorded in μ-law format at 8 KHz requires nearly 0.5 megabytes of storage, while the standard Compact Disc audio format (stereo 16-bit linear PCM data sampled at 44.1 KHz) requires approximately 10 megabytes per minute.

Audio data may be represented in several different formats. An audio device's current audio data format can be determined by using the **AUDIO_GETINFO** **ioctl** described below.

An audio data format is characterized in the audio driver by four parameters: Sample Rate, Encoding, Precision, and Channels. Refer to the device-specific manual pages for a list of the audio formats that each device supports. In addition to the formats that the audio device supports directly, other formats provide higher data compression. Applications may convert audio data to and from these formats when recording or playing.

Sample Rate

Sample rate is a number that represents the sampling frequency (in samples per second) of the audio data.

Encodings

An encoding parameter specifies the audio data representation. μ-law encoding (pronounced mew-law) corresponds to CCITT G.711, and is the standard for voice data used by telephone companies in the United States, Canada, and Japan. A-law encoding is also part of G.711, and is the standard encoding for telephony elsewhere in the world. A-law and μ-law audio data are sampled at a rate of 8000 samples per second with 12-bit precision, with the data compressed to 8-bit samples. The resulting audio data quality is equivalent to that of standard analog telephone service.

Linear Pulse Code Modulation (PCM) is an uncompressed audio format in which sample values are directly proportional to audio signal voltages. Each sample is a 2's complement number that represents a positive or negative amplitude.

Precision Precision indicates the number of bits used to store each audio sample. For instance, µ-law and A-law data are stored with 8-bit precision. PCM data may be stored at various precisions, though 16-bit PCM is most common.

Channels Multiple channels of audio may be interleaved at sample boundaries. A sample frame consists of a single sample from each active channel. For example, a sample frame of stereo 16-bit PCM data consists of 2 16-bit samples, corresponding to the left and right channel data.

DESCRIPTION

The device **/dev/audio** is a device driver that dispatches audio requests to the appropriate underlying audio device driver. The audio driver is implemented as a STREAMS driver. In order to record audio input, applications **open**(2) the **/dev/audio** device and read data from it using the **read**(2) system call. Similarly, sound data is queued to the audio output port by using the **write**(2) system call. Device configuration is performed using the **ioctl**(2) interface.

As some systems may contain more than one audio device, application writers are encouraged to query the **AUDIODEV** environment variable. If this variable is present in the environment, its value should identify the path name of the default audio device.

Opening the Audio Device The audio device is treated as an exclusive resource – only one process can open the device at a time. However, two processes may simultaneously access the device: if one opens it read-only, then another may open it write-only.

When a process cannot open **/dev/audio** because the requested access mode is busy:

- if either the **O_NDELAY** or **O_NONBLOCK** flag are set in the **open( )** *oflag* argument, then −1 is immediately returned, with *errno* set to EBUSY.
- if neither the **O_NDELAY** nor the **O_NONBLOCK** flag are set, then **open( )** hangs until the device is available or a signal is delivered to the process, in which case a −1 is returned with *errno* set to EINTR. This allows a process to block in the **open** call, while waiting for the audio device to become available.

Upon the initial **open( )** of the audio device, the driver will reset the data format of the device to the default state of 8-bit, 8Khz, mono µ-law data. If the device is already open and a different audio format has been set, this will not be possible. Audio applications should explicitly set the encoding characteristics to match the audio data requirements, rather than depend on the default configuration.

Since the audio device grants exclusive read or write access to a single process at a time, long-lived audio applications may choose to close the device when they enter an idle state and reopen it when required. The *play.waiting* and *record.waiting* flags in the audio information structure (see below) provide an indication that another process has requested access to the device. For instance, a background audio output process may choose to relinquish the audio device whenever another process requests write access.

Recording Audio Data The **read( )** system call copies data from the system buffers to the application. Ordinarily, **read( )** blocks until the user buffer is filled. The **I_NREAD ioctl** (see **streamio**(7I)) may be used to determine the amount of data that may be read without blocking. The device may alternatively be set to a non-blocking mode, in which case **read( )** completes immediately, but may return fewer bytes than requested. Refer to the **read**(2) manual page for a complete description of this behavior.

When the audio device is opened with read access, the device driver immediately starts buffering audio input data. Since this consumes system resources, processes that do not record audio data should open the device write-only (**O_WRONLY**).

The transfer of input data to STREAMS buffers may be paused (or resumed) by using the **AUDIO_SETINFO ioctl** to set (or clear) the *record.pause* flag in the audio information structure (see below). All unread input data in the STREAMS queue may be discarded by using the **I_FLUSH** STREAMS **ioctl** (see **streamio**(7I)). When changing record parameters, the input stream should be paused and flushed before the change, and resumed afterward. Otherwise, subsequent reads may return samples in the old format followed by samples in the new format. This is particularly important when new parameters result in a changed sample size.

Input data can accumulate in STREAMS buffers very quickly. At a minimum, it will accumulate at 8000 bytes per second for 8-bit, 8 KHz, mono, μ-law data. If the device is configured for 16-bit linear or higher sample rates, it will accumulate even faster. If the application that consumes the data cannot keep up with this data rate, the STREAMS queue may become full. When this occurs, the *record.error* flag is set in the audio information structure and input sampling ceases until there is room in the input queue for additional data. In such cases, the input data stream contains a discontinuity. For this reason, audio recording applications should open the audio device when they are prepared to begin reading data, rather than at the start of extensive initialization.

Playing Audio Data

The **write( )** system call copies data from an applications buffer to the STREAMS output queue. Ordinarily, **write( )** blocks until the entire user buffer is transferred. The device may alternatively be set to a non-blocking mode, in which case **write( )** completes immediately, but may have transferred fewer bytes than requested (see **write**(2)).

Although **write( )** returns when the data is successfully queued, the actual completion of audio output may take considerably longer. The **AUDIO_DRAIN ioctl** may be issued to allow an application to block until all of the queued output data has been played. Alternatively, a process may request asynchronous notification of output completion by writing a zero-length buffer (end-of-file record) to the output stream. When such a buffer has been processed, the *play.eof* flag in the audio information structure (see below) is incremented.

The final **close**(2) of the file descriptor hangs until audio output has drained. If a signal interrupts the **close( )**, or if the process exits without closing the device, any remaining data queued for audio output is flushed and the device is closed immediately.

The conversion of output data may be paused (or resumed) by using the **AUDIO_SETINFO ioctl** to set (or clear) the *play.pause* flag in the audio information structure. Queued output data may be discarded by using the **I_FLUSH** STREAMS **ioctl**.

Output data will be played from the STREAMS buffers at a rate of at least 8000 bytes per second for μ-law or A-law data (faster for 16-bit linear data or higher sampling rates). If the output queue becomes empty, the *play.error* flag is set in the audio information structure and output is stopped until additional data is written. If an application attempts to write a number of bytes that is not a multiple of the current sample frame size, an error will be generated and the device will need to be closed before any future writes will succeed.

Asynchronous I/O

The **I_SETSIG** STREAMS **ioctl** enables asynchronous notification, through the **SIGPOLL** signal, of input and output ready conditions. The **O_NONBLOCK** flag may be set using the **F_SETFL** **fcntl**(2) to enable non-blocking **read( )** and **write( )** requests. This is normally sufficient for applications to maintain an audio stream in the background.

Audio Control Pseudo-Device

It is sometimes convenient to have an application, such as a volume control panel, modify certain characteristics of the audio device while it is being used by an unrelated process. The **/dev/audioctl** pseudo-device is provided for this purpose. Any number of processes may open **/dev/audioctl** simultaneously. However, **read( )** and **write( )** system calls are ignored by **/dev/audioctl**. The **AUDIO_GETINFO** and **AUDIO_SETINFO** **ioctl** commands may be issued to **/dev/audioctl** to determine the status or alter the behavior of **/dev/audio**. Note: In general, the audio control device name is constructed by appending the letters **"ctl"** to the path name of the audio device.

Audio Status Change Notification

Applications that open the audio control pseudo-device may request asynchronous notification of changes in the state of the audio device by setting the **S_MSG** flag in an **I_SETSIG** STREAMS **ioctl**. Such processes receive a **SIGPOLL** signal when any of the following events occur:

- An **AUDIO_SETINFO** **ioctl** has altered the device state.
- An input overflow or output underflow has occurred.
- An end-of-file record (zero-length buffer) has been processed on output.
- An **open( )** or **close( )** of **/dev/audio** has altered the device state.
- An external event (such as speakerbox volume control) has altered the device state.

IOCTLS

Audio Information Structure

The state of the audio device may be polled or modified using the **AUDIO_GETINFO** and **AUDIO_SETINFO** **ioctl** commands. These commands operate on the **audio_info** structure as defined, in **<sys/audioio.h>**, as follows:

```
/* This structure contains state information for audio device
  IO streams */
struct audio_prinfo {
          /* The following values describe the audio data encoding */
      uint_t    sample_rate;     /* samples per second */
      uint_t    channels;        /* number of interleaved channels */
      uint_t    precision;       /* number of bits per sample */
      uint_t    encoding;        /* data encoding method */
/* The following values control audio device configuration */
      uint_t    gain;            /* volume level */
      uint_t    port;            /* selected I/O port */
      uint_t    buffer_size;     /* I/O buffer size */

/* The following values describe the current device state */
```

```
        uint_t   samples;        /* number of samples converted */
        uint_t   eof;            /* End Of File counter (play only) */
        uchar_t  pause;          /* non-zero if paused, zero to resume */
        uchar_t  error;          /* non-zero if overflow/underflow */
        uchar_t  waiting;        /* non-zero if a process wants access */
        uchar_t  balance;        /* stereo channel balance */

        /* The following values are read-only device state flags */
        uchar_t  open;           /* non-zero if open access granted */
        uchar_t  active;         /* non-zero if I/O active */
        uint_t   avail_ports;    /* available I/O ports */
} audio_prinfo_t;

/* This structure is used in AUDIO_GETINFO and AUDIO_SETINFO ioctl
  commands */
typedef struct audio_info {
        audio_prinfo_t  record;         /* input status information */
        audio_prinfo_t  play;           /* output status information */
        uint_t          monitor_gain;   /* input to output mix */
        uchar_t         output_muted;   /* non-zero if output muted */
}  audio_info_t;

/* Audio encoding types */
#define AUDIO_ENCODING_ULAW      (1)     /* u-law encoding */
#define AUDIO_ENCODING_ALAW      (2)     /* A-law encoding  */
#define AUDIO_ENCODING_LINEAR    (3)     /* Linear PCM encoding */

/* These ranges apply to record, play, and monitor gain values */
#define AUDIO_MIN_GAIN           (0)     /* minimum gain value */
#define AUDIO_MAX_GAIN           (255)   /* maximum gain value */

/* These values apply to the balance field to adjust channel gain values */
#define AUDIO_LEFT_BALANCE       (0)     /* left channel only */
#define AUDIO_MID_BALANCE        (32)    /* equal left/right balance */
#define AUDIO_RIGHT_BALANCE      (64)    /* right channel only */

/* Define some convenient audio port names (for port and avail_ports) */

/* output ports (several might be enabled at once) */
#define AUDIO_SPEAKER            (0x01)  /* output to built-in speaker */
#define AUDIO_HEADPHONE          (0x02)  /* output to headphone jack */
#define AUDIO_LINE_OUT           (0x04)  /* output to line out */

/* input ports (usually only one may be enabled at a time) */
#define AUDIO_MICROPHONE         (0x01)  /* input from microphone */
#define AUDIO_LINE_IN            (0x02)  /* input from line in */

#define                          MAX_AUDIO_DEV_LEN(16)

/* Parameter for the AUDIO_GETDEV ioctl */
typedef struct audio_device {
        char  name[MAX_AUDIO_DEV_LEN];
        char  version[MAX_AUDIO_DEV_LEN];
        char  config[MAX_AUDIO_DEV_LEN];
} audio_device_t;
```

The *play.gain* and *record.gain* fields specify the output and input volume levels. A value of **AUDIO_MAX_GAIN** indicates maximum volume. Audio output may also be temporarily muted by setting a non-zero value in the *output_muted* field. Clearing this field restores audio output to the normal state. Most audio devices allow input data to be monitored by mixing audio input onto the output channel. The *monitor_gain* field controls the level of this feedback path.

The *play.port* field controls the output path for the audio device. It can be set to either **AUDIO_SPEAKER** (built-in speaker), **AUDIO_HEADPHONE** (headphone jack), or **AUDIO_LINE_OUT** (line-out port). For some devices, it may be set to a combination of these ports. The *play.avail_ports* field returns the set of output ports that are currently accessible. The input ports can be either **AUDIO_MICROPHONE** or **AUDIO_LINE_IN**. The *record.avail_ports* field returns the set of input ports that are currently accessible.

The *play.balance* and *record.balance* fields are used to control the volume between the left and right channels when manipulating stereo data. When the value is set between **AUDIO_LEFT_BALANCE** and **AUDIO_MID_BALANCE**, the right channel volume will be reduced in proportion to the *balance* value. Conversely, when *balance* is set between **AUDIO_MID_BALANCE** and **AUDIO_RIGHT_BALANCE**, the left channel will be proportionally reduced.

The *play.pause* and *record.pause* flags may be used to pause and resume the transfer of data between the audio device and the STREAMS buffers. The *play.error* and *record.error* flags indicate that data underflow or overflow has occurred. The *play.active* and *record.active* flags indicate that data transfer is currently active in the corresponding direction.

The *play.open* and *record.open* flags indicate that the device is currently open with the corresponding access permission. The *play.waiting* and *record.waiting* flags provide an indication that a process may be waiting to access the device. These flags are set automatically when a process blocks on **open( )**, though they may also be set using the **AUDIO_SETINFO ioctl** command. They are cleared only when a process relinquishes access by closing the device.

The *play.samples* and *record.samples* fields are initialized, at **open( )**, to zero and increment each time a data sample is copied to or from the associated STREAMS queue. Some audio drivers may be limited to counting buffers of samples, instead of single samples for the *samples* accounting. For this reason, applications should not assume that the *samples* fields contain a perfectly accurate count. The *play.eof* field increments whenever a zero-length output buffer is synchronously processed. Applications may use this field to detect the completion of particular segments of audio output.

The *record.buffer_size* field controls the amount of input data that is buffered in the device driver during record operations. Applications that have particular requirements for low latency should set the value appropriately. Note however that smaller input buffer sizes may result in higher system overhead. The value of this field is specified in bytes and drivers will constrain it to be a multiple of the current sample frame size. Some drivers may place other requirements on the value of this field. Refer to the audio device-specific manual page for more details. If an application changes the format of the audio device and does not modify the *record.buffer_size* field, the device driver may use a default value to compensate for the new data rate. Therefore, if an application wishes to modify this field, it should modify it during or after the format change itself, not before. The *record.buffer_size* field may be modified only on the **/dev/audio** device by processes that have it opened for reading. The *play.buffer_size* field is currently not supported.

The audio data format is indicated by the *sample_rate*, *channels*, *precision*, and *encoding* fields. The values of these fields correspond to the descriptions in the **AUDIO FORMATS** section above. Refer to the audio device-specific manual pages for a list of supported data format combinations.

The data format fields may be modified only on the **/dev/audio** device. The audio hardware will often constrain the input and output data formats to be identical. If this is the case, then the data format may not be changed if multiple processes have opened the audio device.

If the parameter changes requested by an **AUDIO_SETINFO ioctl** cannot all be accommodated, **ioctl( )** will return with *errno* set to EINVAL and no changes will be made to the device state.

Streamio IOCTLS

All of the **streamio**(7I) **ioctl** commands may be issued for the **/dev/audio** device. Because the **/dev/audioctl** device has its own STREAMS queues, most of these commands neither modify nor report the state of **/dev/audio** if issued for the **/dev/audioctl** device. The **I_SETSIG ioctl** may be issued for **/dev/audioctl** to enable the notification of audio status changes, as described above.

Audio IOCTLS

The audio device additionally supports the following **ioctl** commands:

AUDIO_DRAIN
The argument is ignored. This command suspends the calling process until the output STREAMS queue is empty, or until a signal is delivered to the calling process. It may not be issued for the **/dev/audioctl** device. An implicit **AUDIO_DRAIN** is performed on the final **close( )** of **/dev/audio**.

AUDIO_GETDEV
The argument is a pointer to an **audio_device** structure. This command may be issued for either **/dev/audio** or **/dev/audioctl**. The returned value in the *name* field will be a string that will identify the current **/dev/audio** hardware device, the value in *version* will be a string indicating the current version of the hardware, and *config* will be a device-specific string identifying the properties of the audio stream associated with that file descriptor. Refer to the audio device-specific manual pages to determine the actual strings returned by the device driver.

AUDIO_GETINFO
The argument is a pointer to an **audio_info** structure. This command may be issued for either **/dev/audio** or **/dev/audioctl**. The current state of the **/dev/audio** device is returned in the structure.

AUDIO_SETINFO

The argument is a pointer to an **audio_info** structure. This command may be issued for either the **/dev/audio** or the **/dev/audioctl** device with some restrictions. This command configures the audio device according to the structure supplied and overwrites the structure with the new state of the device. Note: The *play.samples*, *record.samples*, *play.error*, *record.error*, and *play.eof* fields are modified to reflect the state of the device when the **AUDIO_SETINFO** was issued. This allows programs to automatically modify these fields while retrieving the previous value.

Certain fields in the information structure, such as the *pause* flags are treated as read-only when **/dev/audio** is not open with the corresponding access permission. Other fields, such as the gain levels and encoding information, may have a restricted set of acceptable values. Applications that attempt to modify such fields should check the returned values to be sure that the corresponding change took effect. The *sample_rate*, *channels*, *precision*, and *encoding* fields treated as read-only for **/dev/audioctl**, so that applications can be guaranteed that the existing audio format will stay in place until they relinquish the audio device. **AUDIO_SETINFO** will return EINVAL when the desired configuration is not possible, or EBUSY when another process has control of the audio device.

Once set, the following values persist through subsequent **open()** and **close()** calls of the device: *play.gain*, *record.gain*, *play.balance*, *record.balance, output_muted*, *monitor_gain*, *play.port*, and *record.port*. However, an automatic device driver unload will reset these parameters to their default values on the next load. All other state is reset when the corresponding I/O stream of **/dev/audio** is closed.

The **audio_info** structure may be initialized through the use of the **AUDIO_INITINFO** macro. This macro sets all fields in the structure to values that are ignored by the **AUDIO_SETINFO** command. For instance, the following code switches the output port from the built-in speaker to the headphone jack without modifying any other audio parameters:

```
audio_info_tinfo;

AUDIO_INITINFO(&info);
info.play.port = AUDIO_HEADPHONE;
err = ioctl(audio_fd, AUDIO_SETINFO, &info);
```

This technique eliminates problems associated with using a sequence of **AUDIO_GETINFO** followed by **AUDIO_SETINFO**.

ERRORS

An **open()** will fail if:

EBUSY The requested play or record access is busy and either the **O_NDELAY** or **O_NONBLOCK** flag was set in the **open()** request.

EINTR The requested play or record access is busy and a signal interrupted the **open()** request.

An **ioctl( )** will fail if:

EINVAL The parameter changes requested in the **AUDIO_SETINFO ioctl** are invalid or are not supported by the device.

EBUSY The parameter changes requested in the **AUDIO_SETINFO ioctl** could not be made because another process has the device open and is using a different format.

FILES

The physcial audio device names are system dependent and are rarely used by programmers. The programmer should use the generic device names listed below.

/dev/audio	symbolic link to the system's primary audio device
/dev/audioctl	symbolic link to the control device for **/dev/audio**
/dev/sound/0	first audio device in the system
/dev/sound/0ctl	audio control device for **/dev/sound/0**

SEE ALSO

open(2), **close**(2), **read**(2), **write**(2), **ioctl**(2), **fcntl**(2), **poll**(2), **audioamd**(7D), **audiocs**(7D), **dbri**(7D), **sbpro**(7D), **streamio**(7I)

BUGS

Due to a *feature* of the STREAMS implementation, programs that are terminated or exit without closing the **audio** device may hang for a short period while audio output drains. In general, programs that produce audio output should catch the **SIGINT** signal and flush the output stream before exiting.

On LX machines running Solaris 2.3, catting a demo audio file to the audio device **/dev/audio** does not work. Use the **audioplay** command on LX machines instead of **cat**.

FUTURE DIRECTIONS

Future workstation audio resources will be managed by an audio foundation library. For the time being, we encourage you to write your programs in a modular fashion, isolating the audio device-specific functions, so that they may be easily ported to such an environment.

The **AUDIO_GETDEV ioctl** is provided for the future implementation of an audio device capability database. In general, applications may use the *play.avail_ports* and *record.avail_ports* fields of the **audio_info** structure to determine the audio device capabilities.