SPARCstation Audio Programming

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: FE318-0
Revision A, July 1991

2550 Garcia Avenue, Mountain View, California 94043-1100

The OPEN LOOK and the Sun Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 (October 1988) and FAR 52.227-19 (June 1987).

The product described in this manual may be protected by one or more U.S. patents, foreign patents, and/or pending applications.

TRADEMARKS

The Sun logo, Sun Microsystems, Sun Workstation, NeWS, and SunLink are registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Sun, Sun-2, Sun-3, Sun-4, Sun386*i*, SunCD, SunInstall, SunOS, SunView, NFS, and OpenWindows are trademarks of Sun Microsystems, Inc.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

PostScript is a registered trademark of Adobe Systems Incorporated. Adobe also owns copyrights related to the PostScript language and the PostScript interpreter. The trademark PostScript is used herein only to refer to material supplied by Adobe or to programs written in the PostScript language as defined by Adobe.

X Window System is a product of the Massachusetts Institute of Technology.

SPARC is a registered trademark of SPARC International, Inc. Products bearing the SPARC trademark are based on an architecture developed by Sun Microsystems, Inc. SPARCstation is a trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc.

All other products or services mentioned in this document are identified by the trademarks, service marks, or product names as designated by the companies who market those products. Inquiries concerning such trademarks should be made directly to those companies.

Contents

Figures

SPARCstation Audio Programming

The desktop SPARCstation™ supports telephone quality audio input and output. Under SunOS™ release 4.1, Sun provides and supports a driver interface to the audio device. It also provides, as demonstration software, a preliminary set of programs that use the audio device, and a library of routines that handles many of the common operations necessary to access and manipulate audio files and devices.

This document is a programmer's guide to the SunOS 4.1 audio interfaces. It presents programming guidelines to help you make sure that applications can easily be ported to future systems. It also provides suggestions to help you use the audio features most effectively.

SPARCstation Audio Characteristics

The SPARCstation audio device plays and records a single channel of sound using the AMD Am79C30A Digital Subscriber Controller™ chip. The chip has a built-in analog to digital converter (ADC) and digital to analog converter (DAC). Digital data is sampled at a rate of 8000 samples per second (8 Khz), with close to 14-bit precision. The data is then compressed, using μ-law encoding, to 8-bit samples. This provides audio data quality equivalent to standard telephone quality audio.

The digital to analog converter can drive either the built-in SPARCstation speaker or an external headphone jack. The audio device circuitry also feeds the input signal back through the output, to let you to monitor the audio input source. The monitor has its own gain control.

Digitization of Audio Data

There are two processes involved in the conversion of an analog sound wave to a digital representation. The two processes are sampling and quantization.

The sampling process, known as Pulse Code Modulation (PCM), determines the height of the waveform at discrete time intervals. The more frequently you sample the waveform (the smaller the time interval) the closer you can approximate the original signal.

The figures below illustrate sampling for a simple waveform. The amplitude of the signal is measured at discrete intervals as indicated by the arrows (Figure 1).

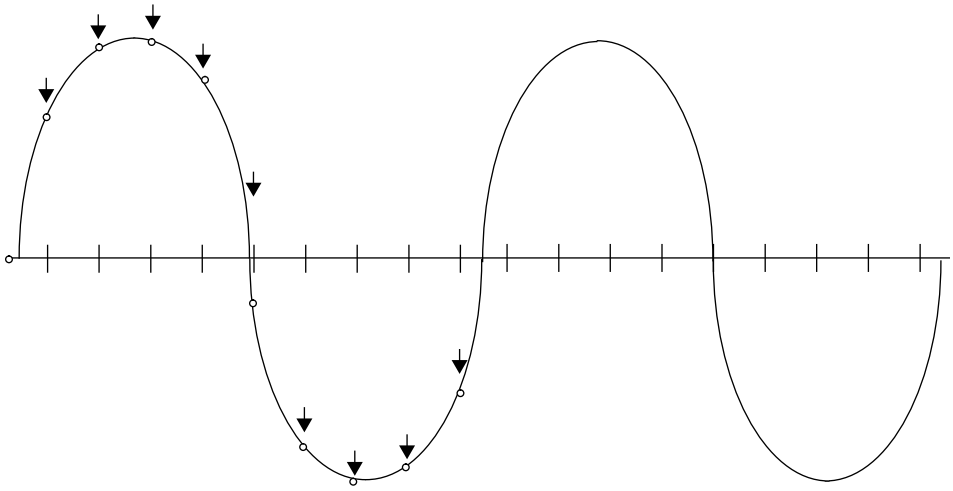

Figure 1 Sample points on a simple waveform

These discrete points can be represented as digital values, and the signal can be approximated from them (Figure 2).

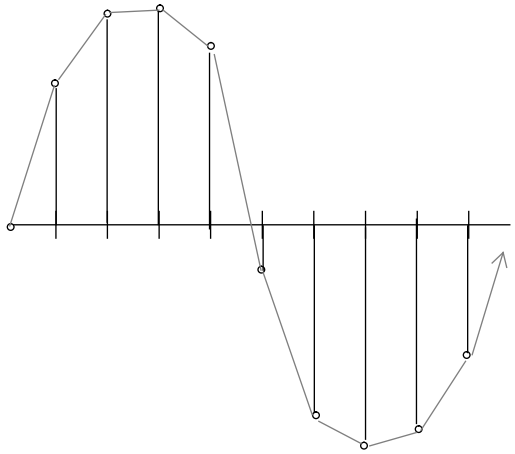

Figure 2 Reproduction of signal from sample points

The Nyquist theorem states that in order to reproduce the original signal, the sampling rate must be greater than twice the bandwidth of the input signal. Sampling at too low a rate produces a phenomenon known as *aliasing*, and the original signal cannot be unambiguously reproduced from the samples. Figure 3 shows this effect. Sampling at twice the frequency still enables reproduction of the signal. However, if the frequency becomes less than 1/2 the sampling rate, the signal cannot be accurately reproduced from the samples.

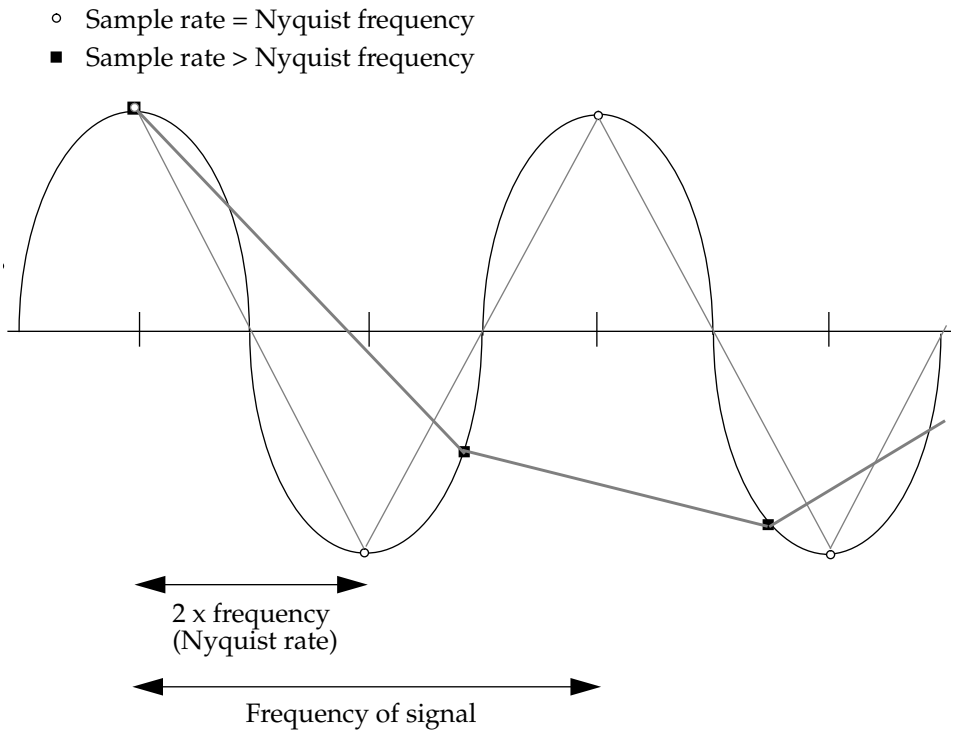

Figure 3 Aliasing and the Nyquist rate

The actual sample rate used by the telephone network is 8000 samples/second. Therefore, the maximum frequency that can be accurately represented is 4000 Hz. Because the bandwidth of today's telephone network voice channel is 3300 Hz, it is easily accommodated. The SPARCstation also samples audio at 8000 samples/second, sufficient for voice-quality audio.

Quantization is a result of the process of digitizing the individual PCM samples. The telephone network (and the SPARCstation) converts each PCM sample into a binary value representing the height of the pulse.

The SPARCstation audio chip has over 16000 possible values available (±8159), which can be represented by just under 14 bits. The ADC assigns to each PCM pulse the one value among the 16000+ that most closely approximates the actual value of the pulse. The difference between the real value and the closest available numeric value is the quantization error. These errors are equivalent

to a small, constant amount of noise, called quantization noise. This noise is characterized by its relationship to the strength of the signal, known as the signal to noise ratio.

The data is then compressed into an 8-bit representation for storage. An additional error (more noise) is introduced by this process.

There are a number of algorithms available that can be used to encode this data. The most straightforward algorithm is linear encoding. Linear encoding allocates the digital approximations of the PCM values in uniform intervals across the entire amplitude range. However in a linear algorithm, the noise generated by the quantization error is as great at low amplitudes as it is at higher amplitudes. Thus the signal to noise *ratio* is much worse at low amplitudes. Because most voice modulation occurs at low amplitude, a linear encoding algorithm will not provide as clean sounding a reproduction of the audio signal as it could, given the number of bits per sample.

To solve this problem, the SPARCstation (and the US and Japanese telephone networks) use a logarithmic encoding scheme known as μ-law. In this encoding algorithm, the spacing of sample intervals is close to linear at low amplitudes, but is closer to logarithmic at high amplitudes. Figure 4 illustrates this difference.

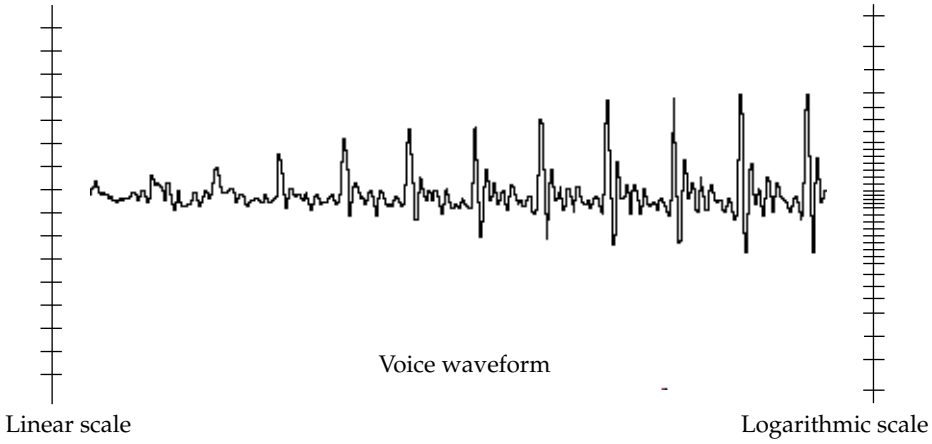

Figure 4 Linear vs. logarithmic spacing of quantization intervals

The quantization error varies with amplitude, but the *ratio* of signal to noise is constant over all amplitudes, and the result sounds better overall. There are also other logarithmic encoding algorithms which can be used for voice data, such as A-law, used by the European telephone network.

Audio Programming Interfaces

Sun currently provides two levels of interface in SunOS 4.1 for the audio device. The first is a fully supported device driver interface. The second is an unsupported library of routines that provides a preliminary, higher-level application programming interface (API) for audio.

Device Driver Interface

SunOS release 4.1 provides a supported driver for the audio device `/dev/audio`. This driver has changed substantially from the preliminary driver released with SunOS release 4.0.3-Sun4c.

The new audio driver is implemented as a STREAMS device. Applications use the standard `open`(2V), `read`(2V), `write`(2V), and `close`(2V) routines to record and play audio data. The interface treats the audio device as an exclusive resource; typically only one process will open the device at a time. However, two processes may access the device simultaneously if one opens it read-only and the other opens it write-only.

The driver interface also provides a pseudo-device, `/dev/audioctl`, which may be opened by any number of processes. This device allows any process to determine the status or alter the behavior of `/dev/audio` while `/dev/audio` is being used by an unrelated process.

Most of the standard `filio`(4) and `streamio`(4) ioctl commands may be issued for the audio device, and the device supports some additional ioctl commands of its own.

The `audio`(4) man page in the *SunOS Reference Manual* (part number 800-3827-10) fully describes this interface.

Most audio programs compiled under release 4.0.3-Sun4c will operate normally in SunOS 4.1; however, they will not compile in the 4.1 environment. Binary compatibility with 4.0.3-Sun4c audio programs will not be maintained in future releases of the operating system. The audio device driver interface provided in SunOS release 4.1 will be supported in future releases.

Preliminary Audio Tools and Application Programming Interface

In addition to the audio device driver interface, there are a number of programs and routines, supplied as demonstration software, that provide interfaces to the audio device. These are available in the `/usr/demo/SOUND` directory.

First, there are two prototype desktop tools that operate in the SunView™ environment.

- Soundtool allows you to record, playback and edit audio files.
- Gaintool is a prototype for an audio control panel that lets you change audio output.

For information on using these two programs, see Appendix C of the *Sun System User's Guide* for your Desktop SPARC® system. You can find additional information in Appendix D of the *SunOS 4.1 Release Manual*.

Note – The SunOS 4.1 version of Soundtool expects audio files to have audio headers, as introduced in the 4.1 preliminary audio API. When Soundtool encounters an audio file without this header, it will issue a warning. Soundtool's store operation will rewrite the file with an audio file header.

A companion filter is also provided, `raw2audio`(6), which converts raw data to the prototype audio file format. This program is useful for converting raw audio data, such as that recorded using the SunOS release 4.0.3c sound demo, the SunOS 4.1 preliminary audio file format.

Second, there are two command-line programs, `play`(6), and`record`(6), that provide the functionality needed to play an audio file, and to record audio data and store it as an audio file. Invoking these programmatically may provide all the functionality necessary for straightforward audio applications.

These programs and routines are furnished on an AS IS basis, to provide a preliminary interface to useful capabilities. They are unsupported, and their syntax or semantics maybe redefined in future releases of SunOS. Copies of manual pages for `soundtool`(6), `gaintool`(6), `play`(6), `record`(6), and `raw2audio`(6) are supplied in `/usr/man/man6`. There is a `README` file in `/usr/demo/SOUND` that provides further information on these programs.

Finally, `/usr/demo/SOUND` contains a library of audio functions, `libaudio.a`, that provides a preliminary interface to many of the common operations necessary to access and manipulate audio files and devices. Manual pages for these routines are provided in `/usr/demo/SOUND/man3`.

Audio Programming Considerations

Typically, an application that uses audio will record audio data and store it in a file, or will play audio data from an existing file. However, raw audio data can be meaningless unless you know its characteristics, such as the sample rate at which it was recorded, the data precision, and the encoding scheme used. Thus, an application program typically will need to associate some attributes about the data along with the audio data itself.

The preliminary audio API provided in `libaudio` addresses these issues. The API provides an audio file format that includes a header structure to handle the relevant data attributes. It also provides routines to manipulate the header, to access and control the audio device, to do transformations between linear and μ-law encodings, and other miscellaneous but useful functions.

The prototype play and record programs, as well as the Soundtool and Gaintool programs, make use of this file structure and interface.

Man pages that describe the preliminary audio file format and API may be found in `/usr/demo/SOUND/man3`.

As a programmer, you will need to decide whether to program using the audio device driver or the preliminary audio API.

The SunOS 4.1 audio device driver reads or writes buffers of data to and from the audio device, but does not deal with how that data is stored in UNIX® files. Thus, an application program that uses the driver-level interface must handle the audio data storage issues itself. In the current audio implementation, an application may get away with making assumptions about

the attributes of audio data, since SPARCstation audio is restricted to one sample rate and data precision. However, it is likely that future Sun systems will not maintain these restrictions.

The advantage in using the driver is that it is fully supported, and will continue to be supported in future versions of the operating system. Thus, code you write now using this interface should continue to function correctly in the future. However, the driver interface may be enhanced in the future to support additional audio capability, such as higher sampling rates, or additional encoding schemes. Changes would be required to the application in order to take advantage of these enhancements.

The most significant drawback to using the driver interface is that it does not address the issues surrounding audio data storage. Keeping track of data attributes such as the sampling rate becomes the responsibility of the application. As a result, audio data stored by one application could be unusable by other applications.

The preliminary audio API was designed to address the drawbacks of the device-level interface. The API provides a generalized mechanism for storing and manipulating audio data. Any application that uses this API should be able to use audio data files from any other application that also uses the API. In addition the interface is designed to allow for future enhancements in the types and characteristics of audio data it can handle. The intention is to minimize the impact of future enhancements on existing applications.

The higher-level routines should simplify the task of the application developer, since much of the work of manipulating the audio data is handled by the routines provided. And by shielding the application from device specific concerns, the impact of low-level changes in the interface can be minimized or eliminated.

The risk in using the API is that it is preliminary, and is not supported by Sun. It is possible that the design of the audio API will change in the future, requiring changes in the application. Thus, any code for audio should be isolated so that it can be changed easily. However, the intent of the higher-level API is to minimize the amount of changes that may be necessary, compared to what would be required to support new features using the device driver interface.

Despite the fact that it is preliminary, the API provides the most flexible interface for audio applications. Audio programmers should use either the library interface, or the play and record routines whenever possible. As much as possible, future audio capabilities will be implemented underneath these interfaces in a transparent way. The more removed your application is from device details, the less likely it is to be impacted by low-level changes.

Preliminary Audio File Format

The preliminary audio library provides routines to store and retrieve a header structure for audio data files. This header contains the attributes necessary to characterize the audio data.

The format of the audio file header is compatible with a subset of the NeXT basic sound file header. This format should allow the transparent sharing of audio files in a networked environment.

The file header structure includes a "magic number" that identifies the file as an audio file. This is followed by a number of fields that describe attributes of the data such as the sample rate, number of samples per unit, number of channels, and the encoding format used to store the data.

The audio file format also provides a variable length information field whose contents can be defined by the application. For example, the application developer may want to store ASCII annotations about the source of the audio data (artist's name, description, song title, date recorded).

Application programs access the data description fields through the in-core header structure. The in-core header format is not an exact duplicate of the file header format. The preliminary audio library provides routines to determine if a file is an audio file, to read the audio file header into the in-core audio header structure, and to write the header out to a file. This means that application programs don't need to be concerned with the details of the file header structure, and may be insulated from possible future changes in the structure. The audio header structure and the library routines are described further later in this document.

SPARCstation audio currently supports only one sample rate, one data precision and encoding type and a single channel; however, the header structure has been defined to allow more variation in the future. It also will handle audio data from other sources that may have different characteristics.

Both the header and audio data fields are written using *big-endian* byte ordering. This is to facilitate transparent file access across multiple machine architectures.

Programming Audio Using `play()` *and* `record()`

The most straightforward use of audio is to play sounds from stored files, or record sounds into a file. For example, an application might allow a user to record comments into a file that is saved along with the work he's doing. Then at a later time he can play back his earlier comments. An application of this type could use the `play` and `record` routines to accomplish this.

The `play` program simply plays the contents of an audio file you specify. `record` takes in audio and puts it into a file you specify. These routines do the file and device manipulation and error checking for you. However, the application will need to provide any user interface that may be necessary, such as a way to pause the recording process.

Both these programs use the audio file format defined in the preliminary audio API.

The `play` program copies the contents of named audio files to the audio device. If you do not provide a file name, `play` will read from `stdin`. In either case, the input stream must contain a valid audio file header. The encoding information is compared with the capabilities of the audio device. If the data is incompatible with the device, the routine will print an error and skip that file.

Normally, if the audio device is not available, `play` will wait until it can obtain access to the device. However, you can use a command line option to have `play` exit immediately if the device is busy.

The `play` program also provides options to control the output volume level, and to specify an alternate audio device. `/dev/audio` is the default audio device when no option is specified.

The play command and its options are described in detail in the `play`(6) man page.

The `record` program copies audio data from the audio device and stores it into a named audio file. This file will be prefixed by an audio file header, whose encoding information is taken from the characteristics of the audio

device. Recording begins immediately and continues until a `SIGINT` signal (such as `^C`) is received. Optionally, recording can continue until a specified quantity of data has been recorded. If the audio device is unavailable for reading, the record program will print an error to STDERR and exit.

An option lets you specify that data should be appended onto the end of an existing audio file. In this case the encoding of the file must match the characteristics of the device configuration, unless you override the restriction with a Force flag.

Other options let you set the recording gain; specify a string to be stored in the information field of the output file header; and specify a maximum length of time to record.

The details of these options and flags are discussed in the `record`(6) man page.

Any settings a user specifies using the Gaintool program, such as switching the output port between headphones and built-in speaker, *will* affect the `record` or `play` programs.

To invoke these programs from an application, you should `fork`(2) a new process and `exec` the program. This allows you to wait for their completion asynchronously. You should NOT use the `system`(3) function to run these programs.

Programming Audio Using `libaudio`

If you want more control over the audio I/O process, your program can input and output directly to the audio device. You use the normal UNIX file I/O routines, `open`(2V), `read`(2V) and/or `write`(2V).

In addition, the preliminary audio library provides a set of useful routines that simplify the tasks associated with audio I/O. These include routines for managing the audio device (instead of `ioctl` commands), and to interface to the special file format for audio data.

To use the preliminary API routines, your code needs to do the following:

- Check the status of the audio device to determine whether it is available
- Open the audio device
- Open and obtain the file descriptor for the audio data file
- Access the audio header

- Verify that the characteristics of the audio data in the file (such as the sample rate or encoding scheme) match the characteristics of the device

After completing these steps, you can read data from or write data to the device.

The AUDIO(4) manual page contains a great deal of valuable information about the audio device and using it for audio I/O. You should become familiar with the information there before you begin to program your audio application.

If you are recording data, when you finish you will need to make sure the audio header contains the appropriate information, and then write it into the data file.

You can look at the C source code for the `play` and `record` programs (in `/usr/demo/SOUND`) as a simple example of how to use the API routines. You can find an overview of the audio library in the `audio_intro`(3) man page.

To use these routines, you must include the header files:

```
#include  <multimedia/libaudio.h>
#include  <multimedia/audio_device.h>
```

The following sections briefly describe the routines provided in `libaudio.`

Routines for Managing the Audio File Headers

These routines discussed in this section are documented in detail in the `audio_filehdr(3)` man page.

`audio_isaudiofile()` lets you quickly determine if a file is an audio file. It does limited consistency checking, and returns `TRUE` if the named file appears to be an audio file.

`audio_read_filehdr()` reads the header from an audio file, decodes it and places it into an in-core structure so that it can be manipulated by an application. The routine updates the current file position of the stream to refer to the beginning of the audio data, and sets the data size field in the header to the length, in bytes, of the data. If the length cannot be determined, (for instance, if the data is being read from a pipe) the data size field will be set to the special value `AUDIO_UNKNOWN_SIZE`. `audio_read_filehdr()` also copies the optional information field into a buffer, if you specify one.

This routine does more extensive consistency checking than `audio_isaudiofile()`.

`audio_write_filehdr()` takes the in-core header structure, encodes it and writes it out to the specified file (output stream). It also optionally writes the information buffer immediately following the file header.

`audio_rewrite_filesize()` rewrites the audio file header, setting the data length to a newly specified size. You might need to use this routine if you appended data to an existing audio file.

Application programs access the data description fields through the in-core header structure. The in-core header format is not an exact duplicate of the file header format. The fields in the in-core structure include:

- the sample rate (samples per second)
- number of samples per unit
- number of bytes per unit
- number of channels
- data encoding format
- length of data (advisory only)

The sample rate represents the sampling frequency of the audio data. Currently the only sample rate the SPARCstation supports is 8000 samples per second.

The number of samples per unit and the bytes per unit together describe an individual sound unit. Some encoding schemes combine several samples into a single sound unit that must be treated as an atomic entity. However for PCM and related encodings, the samples per unit is always one. With current SPARCstation audio, there is one byte per sample.

The number of channels describes the number of interleaved audio channels included in the data. For current SPARCstation audio this value will be 1, as the SPARCstation currently supports only one channel.

The data encoding format field enumerates the specific data encoding. Since the SPARCstation currently supports only μ-law, this field will have the value `AUDIO_ENCODING_ULAW.`

The length of data field indicates the number of bytes in the data stream, if it can be determined.

The in-core header structure is defined in <multimedia/audio_hdr.h>. Detailed information on the structure and the definition of these fields can be found in the audio_hdr(3) man page.

The library also provides a routine, audio_decode_filehdr(), which decodes a buffer presumed to contain an audio file header. However, applications should normally use audio_read_filehdr(), which decodes the header as it reads it from the input stream. Applications should avoid dealing with the audio file header directly. This is a prototype implementation, and the details might change in the future. By using the routines provided to access the header, your application will be insulated from any such changes.

Device Control Routines

The following routines enable applications to query and control the state of the audio device without knowing details of the audio ioctl commands. Using these routines will insulate your application from the details of the device.

All of these functions take the file descriptor of the audio device as the argument. In most cases, the file descriptor may refer either to the audio I/O device (/dev/audio) or the audio control pseudo-device (/dev/audioctl).

Play/Resume

audio_pause_play() and audio_pause_record() pause the output or input processes, respectively. audio_pause() pauses both input and output simultaneously.

audio_resume_play(), audio_resume_record(), and audio_resume() resume the paused processes.

Flush Queues

The functions audio_flush_play(), audio_flush_record(), and audio_flush() flush the appropriate audio STREAMS buffers (input queue, output queue, or both simultaneously).

Synchronization

`audio_drain()` causes the requesting process to block until all previously queued output data has been played.

audio_play_eof() writes an End-Of-File marker to the device.

Gain

The following routines manipulate the various gains associated with the audio device. The input, output, and monitor signals each have their own, independent gain settings. The monitor is the input signal fed back through the output, to let you listen to the audio input source.

`audio_get_play_gain()`, `audio_get_record_gain()`, and `audio_get_monitor_gain` retrieve the current gain settings for output, input or the monitor. `audio_set_play_gain()`. `audio_set_record_gain()`, and `audio_set_monitor_gain()` set the respective gain levels. The gain is specified as a floating point value between 0 (infinite attenuation) and 1 (maximum gain of the device).

Device Status

There are a large number of routines that interrogate and set various states or characteristics of the audio device. These take as arguments both a file descriptor and the address of an unsigned integer that contains the value of the particular state field.

The `audio_get_play_port()` and `audio_get_record_port()` functions retrieve the current configuration of the output or input port. `audio_set_play_port()` and `audio_set_record_port()` set the output or input port configuration, and then return the current value. Valid values for the play port are `AUDIO_SPEAKER` or `AUDIO_HEADPHONE`.

`audio_get_play_samples()` and `audio_get_record_samples()` retrieve the current output or input sample count. The `audio_set_play_samples()` and `audio_set_record_samples()` functions set the output or input sample counter. The set functions return the value of the sample counter as it was before the new value is set.

`audio_get_play_error()` sets the value of the state argument to `TRUE` if the output overflow indicator is set. `audio_get_record_error()` does the same if the input underflow indicator is set. The routines

`audio_set_play_error()` and `audio_set_record_error()` set the output or input error flags. Normally, you should use these routines only to reset the error flag to zero. The set functions return the value of the error flag as it was before being set to the new value.

`audio_get_play_eof()` retrieves the current output end-of-file counter. `audio_set_play_eof()` sets the output end-of-file counter. It returns the value of the counter as it was immediately prior to being set with the new value.

`audio_get_play_open()` and `audio_get_record_open()` set the argument `TRUE` if the respective device is open for input or output.

`audio_get_play_active()` and `audio_get_record_active()` set the argument `TRUE` if data output or input is currently in progress.

`audio_get_play_waiting()` and `audio_get_record_waiting()` return the argument `TRUE` if a process is waiting for write or read access to the audio device. `audio_set_play_waiting()` and `audio_set_record_waiting()` allow you to set the output or input waiting flag to `TRUE`, to indicate a process is waiting. The waiting flag will be cleared when the device becomes available for the requested I/O function. You cannot clear the waiting flag using these functions.

Encoding Configuration

The following two routines store current audio device configuration information into an in-core audio header structure. You can then write this structure to a file using the `audio_write_filehdr()` function. The `Audio_hdr` structure is defined in `<multimedia/audio_hdr.h>`, and in the `AUDIO_HDR`(3) man page.

`audio_get_play_config()` retrieves the current output configuration and puts it into the specified `Audio_hdr` structure. `audio_get_record_config()` does the same with the current input configuration.

`audio_set_play_config()` and `audio_set_record_config()` attempt to set the device to match the settings in the given `Audio_hdr` structure. The new configuration is returned.

Direct Device Control

There may be certain cases where you wish to read or modify device state fields directly. The following functions allow this. However, you should avoid using them as much as possible, since they expose the audio ioctl interface to the application.

As arguments, these functions take both a file descriptor for the audio device, and the address of an `Audio_info_t` structure. To use these routines, you must include the header file

```
#include  <sun/audioio.h>
```

The routine `audio_getinfo()` returns the current device information structure. The routine `audio_setinfo()` sets the device state to reflect the values in the specified `Audio_info_t` structure.

The `Audio_info_t` structure will be discussed further in this paper in the section on programming audio using the device driver. Details on this structure and the device state fields are included in the `AUDIO(4)` man page.

Data Format Conversions

Audio signal processing is simply the performance of arithmetic operations on linear data. For example, mixing two sounds is a simple addition of two linear signals. Amplitude scaling is multiplication. However, the SPARCstation audio chip processes data that is encoded using μ-law companding, which is not linear. The μ-law transfer function results in a nearly linear relationship to PCM at low amplitudes, but a logarithmic relationship at high amplitudes. Performing arithmetic operations directly on μ-law encoded data would give incorrect results. You must first convert your data to linear PCM, then perform the operations, and then convert it back to μ-law.

The following routines are provided to convert μ-law data to and from signed integers of various precisions.

The three routines `audio_u2c()`, `audio_u2s()` and `audio_u2l()` convert μ-law data to PCM signed integers of 8, 16, or 32 bits respectively. The routines `audio_c2u()`, `audio_s2u()` and `audio_l2u()` convert from PCM signed integers to their μ-law encoded forms.

As an example, to sum two μ-law values, you might use these routines as follows:

```
sum = audio_s2u(audio_u2s(ulaw1) + audio_u2s(ulaw2));
```

These routines are documented in the man page AUDIO_ULAW2LINEAR(3).

The library also includes routines that facilitate conversion between integer and floating-point PCM. `audio_c2d()`, `audio_s2d()`, and `audio_l2d()` convert 8, 16 or 32-bit PCM signed integers to floating-point values that range between +1 and -1. The routines `audio_d2c()`, `audio_d2s()` and `audio_d2l()` convert floating point data to PCM signed integers of 8, 16 or 32-bit precision. These routines are documented in more detail in the AUDIO_CONVERT(3) man page.

Miscellaneous Library Functions

The audio library contains a number of miscellaneous functions that perform some simple but useful transformations. These functions are documented in the AUDIO_MISC(3) man page.

`audio_bytes_to_secs()` converts a byte count into a floating point time value that is appropriate to the encoding described by the specified audio header. The routine takes the address of an audio header and a byte count as its arguments. The time value indicates the number of seconds of audio that correspond to the given number of bytes of data. `audio_secs_to_bytes()` converts a floating-point time value to a byte count, rounded down to a sample frame boundary.

`audio_str_to_secs()` converts an ASCII string of the form `[hh:][mm:][ss][.dd]` into a floating-point time value. `audio_secs_to_str()` converts a floating-point time value to an ASCII string.

The routine `audio_cmp_hdr()` compares the encoding information fields of tow specified audio headers. The routine returns `0` if the headers are identical, `1` if the only sample rate is different, and `-1` if there are other differences.

`audio_enc_to_str()` converts the encoding information in a given audio header to a printable ASCII string.

Programming Audio Using the Device Driver

The audio device driver is implemented as a STREAMS device. In order to record audio input, an application opens (`open`(2V)) the audio device `/dev/audio`, and reads from it using the `read`(2V) system call. An application queues sound data to the audio output port using the `write`(2V) system call. This is the same, whether using the device driver interface or using the preliminary audio API discussed previously. However, instead of using the routines in libaudio to control the device, you can use a series of ioctl commands to set or interrogate the audio device (or the audio pseudo-device).

Detailed information on the audio device and audio I/O is contained in the `AUDIO`(4) manual page.

/dev/audio and /dev/audioctl

Certain applications, such as a volume control panel, may need to modify characteristics of the audio device while it is being used by an unrelated process. For example, while one application is playing a sound file, you may want a separate control panel application to be able to control the gain (volume) or speaker port (e.g. switch the output to headphones). The pseudo-device `/dev/audioctl` is provided for this purpose.

The `/dev/audio` device may be opened by at most two processes; one for reading, one for writing. `/dev/audioctl` can be opened by any number of processes. `read`(2V) and `write`(2V) system calls are ignored by `/dev/audioctl`. However, the `AUDIO_GETINFO` and `AUDIO_SETINFO` ioctl commands can be issued to `/dev/audioctl` to determine the status or alter the behavior of `/dev/audio`.

Audio Device Information Structure

The audio device information structure is defined in `<sun/audioio.h>`, and described in the `AUDIO`(4) man page. The structure contains separate information structures for both input and output. Each structure contains values describing the audio data encoding (sample rate, number of channels, precision, and encoding method); audio device configuration (gain, selected I/O part); current device state (number of samples converted, end-of-file counter, and flags to indicate if the device is paused, if overflow/underflow has occurred, and if there is a process waiting for access); and read-only device

state flags (indicating if open access has been granted, and if the device is active). The meaning and values of the various flags and fields are discussed in detail in the man pages.

IOCTL Commands

All of the `fileio`(4) and `streamio`(4) ioctl commands may be issued for the `/dev/audio` device. Because `/dev/audioctl` has its own STREAMS queues, most of these commands do not affect or report the state of `/dev/audio` if issued for `/dev/audioctl`. The `I_SETSIG` ioctl may be issued for `/dev/audioctl` to enable the notification of audio status changes.

The audio device supports the following ioctl commands.

`AUDIO_GETINFO` returns the current state of the `/dev/audio` device in a specified `audio_info` structure. This command may be issued for either `/dev/audio` or `/dev/audioctl`.

`AUDIO_SETINFO` configures the audio device according to the contents of the `audio_info` structure supplied as an argument to command. It overwrites the structure with the new state of the device. The command may be issued for either `/dev/audio` or `/dev/audioctl`.

You can initialize the `audio_info` structure with the `AUDIO_INITINFO` macro. This macro sets all field in the structure to values that will be ignored by the `AUDIO_SETINFO` command. This means that once you have initialized the structure with `AUDIO_INITINFO`, you can use `AUDIO_SETINFO` to modify individual settings without affecting any other parameters.

Finally, the `AUDIO_DRAIN` ioctl command suspends the calling process until the output STREAMS queue is empty, or until a signal is delivered to the calling process. This command can only be issued for `/dev/audio`.

Connecting Microphones and External Speakers

SPARCstations are now shipped with a cable that has a 1/8″ mini-phone jack for input, and a 1/8″ mini-phone jack for output. If you do not have this cable, you can order it from Sun. The part number is 530-1594-01. The cable is illustrated in Figure 5.