## NAME

libsox − SoX, an audio file-format and effect library

## SYNOPSIS

**#include <sox.h>**

**int sox_format_init(void);**

**void sox_format_quit(void);**

**sox_format_t sox_open_read(const char *_path_, const sox_signalinfo_t *_info_, const char *_filetype_);**

**sox_format_t sox_open_write(sox_bool (*_overwrite_permitted_)(const char *_filename_), const char *_path_, const sox_sig**

**sox_size_t sox_read(sox_format_t _ft_, sox_ssample_t *_buf_, sox_size_t _len_);**

**sox_size_t sox_write(sox_format_t _ft_, sox_ssample_t *_buf_, sox_size_t _len_);**

**int sox_close(sox_format_t _ft_);**

**int sox_seek(sox_format_t _ft_, sox_size_t _offset_, int _whence_);**

**sox_effect_handler_t const *sox_find_effect(char const *_name_);**

**sox_effect_t *sox_create_effect(sox_effect_handler_t const *_eh_);**

**int sox_effect_options(sox_effect_t *_effp_, int _argc_, char * const _argv[]_);**

**sox_effects_chain_t *sox_create_effects_chain(sox_encodinginfo_t const *_in_enc_, sox_encodinginfo_t const *_out_en**

**void sox_delete_effects_chain(sox_effects_chain_t *_ecp_);**

**int sox_add_effect(sox_effects_chaint_t *_chain_, sox_effect_t*_effp_, sox_signalinfo_t *_in_, sox_signalinfo-t const *_out_);**

**cc** *file.c* **-o** *file* **-lsox**

## DESCRIPTION

*libsox* is a library of sound sample file format readers/writers and sound effects processors. It is mainly developed for use by SoX but is useful for any sound application.

**sox_format_init** function performs some required initialization related to all file format handlers. If compiled with dynamic library support then this will detect and initialize all external libraries. This should be called before any other file operations are performed.

**sox_format_quit** function performs some required cleanup related to all file format handlers.

**sox_open_input** function opens the file for reading whose name is the string pointed to by *path* and associates an sox_format_t with it. If *info* is non-NULL then it will be used to specify the data format of the input file. This is normally only needed for headerless audio files since the information is not stored in the file. If *filetype* is non-NULL then it will be used to specify the file type. If this is not specified then the file type is attempted to be derived by looking at the file header and/or the filename extension. A special name of "-" can be used to read data from stdin.

**sox_open_output** function opens the file for writing whose name is the string pointed to by *path* and associates an sox_format_t with it. If *info* is non-NULL then it will be used to specify the data format of the output file. Since most file formats can write data in different data formats, this generally has to be specified. The info structure from the input format handler can be specified to copy data over in the same format. If *comment* is non-NULL, it will be written in the file header for formats that support comments. If *filetype* is non-NULL then it will be used to specify the file type. If this is not specified then the file type is attempted to be derived by looking at the filename extension. A special name of "-" can be used to write data to stdout.

The function **sox_read** reads *len* samples in to *buf* using the format handler specified by *ft*. All data read is converted to 32-bit signed samples before being placed in to *buf*. The value of *len* is specified in total samples. If its value is not evenly divisable by the number of channels, undefined behavior will occur.

The function **sox_write** writes *len* samples from *buf* using the format handler specified by *ft*. Data in *buf* must be 32-bit signed samples and will be converted during the write process. The value of *len* is specified

in total samples. If its value is not evenly divisable by the number of channels, undefined behavior will occur.

The **sox_close** function dissociates the named *sox_format_t* from its underlying file or set of functions. If the format handler was being used for output, any buffered data is written first.

The function **sox_find_effect** finds effect *name*, returning a pointer to its *sox_effect_handler_t* if it exists, and NULL otherwise.

The function **sox_create_effect** instantiates an effect into a *sox_effect_t* given a *sox_effect_handler_t* *. Any missing methods are automatically set to the corresponding **nothing** method.

The function **sox_effect_options** allows passing options into the effect to control its behavior. It will return SOX_EOF if there were any invalid options passed in. On success, the *effp->in_signal* will optional contain the rate and channel count it requires input data from and *effp->out_signal* will optionally contain the rate and channel count it outputs in. When present, this information should be used to make sure appropriate effects are placed in the effects chain to handle any needed conversions.

Passing in options is currently only supported when they are passed in before the effect is ever started. The behavior is undefined if its called once the effect is started.

**sox_create_effects_chain** will instantiate an effects chain that effects can be added to. *in_enc* and *out_enc* are the signal encoding of the input and output of the chain respectively. The pointers to *in_enc* and *out_enc* are stored internally and so their memory should not be freed. Also, it is OK if their values change over time to reflect new input or output encodings as they are referenced only as effects start up or are restarted.

**sox_delete_effects_chain** will release any resources reserved during the creation of the chain. This will also call **sox_delete_effects** if any effects are still in the chain.

**sox_add_effect** adds an effect to the chain. *in* specifies the input signal info for this effect. *out* is a suggestion as to what the output signal should be but depending on the effects given options and on *in* the effect can choose to do differently. Whatever output rate and channels the effect does produce are written back to *in*. It is meant that *in* be stored and passed to each new call to **sox_add_effect** so that changes will be propagated to each new effect.

SoX includes skeleton C files to assist you in writing new formats (skelform.c) and effects (skeleff.c). Note that new formats can often just deal with the header and then use raw.c's routines for reading and writing.

example0.c and example1.c are a good starting point to see how to write applications using libsox. sox.c itself is also a good reference.

## RETURN VALUE

Upon successful completion **sox_open_input** and **sox_open_output** return an *sox_format_t* (which is a pointer). Otherwise, NULL is returned. TODO: Need a way to return reason for failures. Currently, relies on **sox_warn** to print information.

**sox_read** and **sox_write** return the number of samples successfully read or written. If an error occurs, or the end-of-file is reached, the return value is a short item count or SOX_EOF. TODO: **sox_read** does not distiguish between end-of-file and error. Need an feof() and ferror() concept to determine which occured.

Upon successful completion **sox_close** returns 0. Otherwise, SOX_EOF is returned. In either case, any further access (including another call to **sox_close**()) to the handler results in undefined behavior. TODO: Need a way to return reason for failures. Currently, relies on sox_warn to print information.

Upon successful completion **sox_seek** returns 0. Otherwise, SOX_EOF is returned. TODO Need to set a global error and implement sox_tell.

## ERRORS

TODO

## INTERNALS

SoX's formats and effects operate with an internal sample format of signed 32-bit integer. The data processing routines are called with buffers of these samples, and buffer sizes which refer to the number of samples processed, not the number of bytes. File readers translate the input samples to signed 32-bit integers and return the number of samples read. For example, data in linear signed byte format is left-shifted 24 bits.

Representing samples as integers can cause problems when processing the audio. For example, if an effect to mix down left and right channels into one monophonic channel were to use the line

```
*obuf++ = (*ibuf++ + *ibuf++)/2;
```

distortion might occur since the intermediate addition can overflow 32 bits. The line

```
*obuf++ = *ibuf++/2 + *ibuf++/2;
```

would get round the overflow problem (at the expense of the least significant bit).

Stereo data is stored with the left and right speaker data in successive samples. Quadraphonic data is stored in this order: left front, right front, left rear, right rear.

## FORMATS

A *format* is responsible for translating between sound sample files and an internal buffer. The internal buffer is store in signed longs with a fixed sampling rate. The *format* operates from two data structures: a format structure, and a private structure.

The format structure contains a list of control parameters for the sample: sampling rate, data size (8, 16, or 32 bits), encoding (unsigned, signed, floating point, etc.), number of sound channels. It also contains other state information: whether the sample file needs to be byte-swapped, whether sox_seek() will work, its suffix, its file stream pointer, its *format* pointer, and the *private* structure for the *format* .

The *private* area is just a preallocated data array for the *format* to use however it wishes. It should have a defined data structure and cast the array to that structure. See voc.c for the use of a private data area. Voc.c has to track the number of samples it writes and when finishing, seek back to the beginning of the file and write it out. The private area is not very large. The ''echo'' effect has to malloc() a much larger area for its delay line buffers.

A *format* has 6 routines:

| | |
|---|---|
| startread | Set up the format parameters, or read in a data header, or do what needs to be done. |
| read | Given a buffer and a length: read up to that many samples, transform them into signed long integers, and copy them into the buffer. Return the number of samples actually read. |
| stopread | Do what needs to be done. |
| startwrite | Set up the format parameters, or write out a data header, or do what needs to be done. |
| write | Given a buffer and a length: copy that many samples out of the buffer, convert them from signed longs to the appropriate data, and write them to the file. If it can't write out all the samples, fail. |
| stopwrite | Fix up any file header, or do what needs to be done. |

## EFFECTS

Each effect runs with one input and one output stream. An effect's implementation comprises six functions that may be called to the follow flow diagram:

```
LOOP (invocations with different parameters)
  getopts
  LOOP (invocations with the same parameters)
    LOOP (channels)
      start
    LOOP (whilst there is input audio to process)
```

```
      LOOP (channels)
        flow
    LOOP (whilst there is output audio to generate)
      LOOP (channels)
        drain
    LOOP (channels)
      stop
  kill
```

Notes: For some effects, some of the functions may not be needed and can be NULL. An effect that is marked 'MCHAN' does not use the LOOP (channels) lines and must therefore perform multiple channel processing inside the affected functions. Multiple effect instances may be processed (according to the above flow diagram) in parallel.

| | |
|---|---|
| getopts | is called with a character string argument list for the effect. |
| start | is called with the signal parameters for the input and output streams. |
| flow | is called with input and output data buffers, and (by reference) the input and output data buffer sizes. It processes the input buffer into the output buffer, and sets the size variables to the numbers of samples actually processed. It is under no obligation to read from the input buffer or write to the output buffer during the same call. If the call returns SOX_EOF then this should be used as an indication that this effect will no longer read any data and can be used to switch to drain mode sooner. |
| drain | is called after there are no more input data samples. If the effect wishes to generate more data samples it copies the generated data into a given buffer and returns the number of samples generated. If it fills the buffer, it will be called again, etc. The echo effect uses this to fade away. |
| stop | is called when there are no more input samples and no more output samples to process. It is typically used to release or close resources (e.g. allocated memory or temporary files) that were set-up in *start*. See echo.c for an example. |
| kill | is called to allow resources allocated by *getopts* to be released. See pad.c for an example. |

## LINKING

The method of linking against libsox depends on how SoX was built on your system. For a static build, just link against the libraries as normal. For a dynamic build, you should use libtool to link with the correct linker flags. See the libtool manual for details; basically, you use it as:

libtool --mode=link gcc -o prog /path/to/libsox.la

## BUGS

This manual page is both incomplete and out of date.

## SEE ALSO

**sox**(1), **soxformat**(7)

example*.c in the SoX source distribution.

## LICENSE

Copyright 1998−2009 by Chris Bagwell and SoX Contributors.
Copyright 1991 Lance Norskog and Sundry Contributors.

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See

the GNU Lesser General Public License for more details.

## AUTHORS

Chris Bagwell (cbagwell@users.sourceforge.net). Other authors and contributors are listed in the ChangeLog file that is distributed with the source code.