NAME

libsox_ng, another audio file format and effect library

SYNOPSIS

#include <sox_ng.h>

int sox_format_init(void);

void sox_format_quit(void);

sox_format_t *sox_open_read(char const *_path_, sox_signalinfo_t const *_signal_, sox_encodinginfo_t const *_encoding_, char const *_filetype_);

sox_format_t *sox_open_write(char const *_filename_, sox_signalinfo_t const *_signal_, sox_encodinginfo_t const *_encoding_, char const *_filetype_, sox_oob_t const *_oob_, sox_bool (*overwrite_permitted)(char const *_filename_));

size_t sox_read(sox_format_t *_format_, sox_sample_t *_buf_, size_t _len_);

size_t sox_write(sox_format_t *_format_, sox_sample_t const *_buf_, size_t _len_);

int sox_seek(sox_format_t *_format_, size_t _offset_, int _whence_);

int sox_close(sox_format_t *_format_);

sox_effect_handler_t const *sox_find_effect(char const *_name_);

sox_effect_t *sox_create_effect(sox_effect_handler_t const *_effect_);

int sox_effect_options(sox_effect_t *_effect_, int _argc_, char * const _argv_[]);

sox_effects_chain_t *sox_create_effects_chain(sox_encodinginfo_t const *_in_enc_, sox_encodinginfo_t const *_out_enc_);

int sox_add_effect(sox_effects_chain_t *_chain_, sox_effect_t *_effect_, sox_signalinfo_t *_in_, sox_signalinfo_t const *_out_);

void sox_delete_effects_chain(sox_effects_chain_t *_ecp_);

cc *file.c* **-o** *file* **-lsox_ng**

TYPES

The following structures describe audio file formats and how they encode samples:

```
typedef struct sox_format_t {
  char         *filename;       /* File name                          */
  sox_signalinfo_t   signal;    /* Signal specifications              */
  sox_encodinginfo_t encoding;  /* Encoding/decoding specifications */
  char         *filetype;       /* Type of file, "wav" etc.           */
  sox_oob_t    oob;             /* Out-of-band data                   */
  sox_bool     seekable;        /* Can we seek in this file?          */
  char         mode;            /* Read or write mode ('r' or 'w')  */
  sox_uint64_t olength;         /* Samples * chans written to file  */
  sox_uint64_t clips;           /* Incremented if clipping occurs   */
  int          sox_errno;       /* Failure error code                 */
  char         sox_errstr[256]; /* Failure error text                 */
  ...
} sox_format_t;

typedef double sox_rate_t;      /* sample frames per second           */

typedef struct sox_signalinfo_t {
  sox_rate_t   rate;      /* samples per second, 0 if unknown       */
  unsigned     channels;  /* number of sound channels, 0 if unknown */
  unsigned     precision; /* bits per sample, 0 if unknown          */
```

```
  sox_uint64_t length;    /* samples * chans in file, 0 if unknown  */
  ...
} sox_signalinfo_t;

typedef struct sox_encodinginfo_t {
  sox_encoding_t encoding;      /* format of sample numbers         */
  unsigned bits_per_sample;     /* 0 if unknown or variable;
                                 * uncompressed value if lossless;
                                 * compressed value if lossy        */
  double compression;           /* compression factor if applicable */
  sox_option_t reverse_bytes;   /* should bytes be reversed?        */
  sox_option_t reverse_nibbles; /* should nibbles be reversed?      */
  sox_option_t reverse_bits;    /* should bits be reversed?         */
  sox_bool opposite_endian;     /* reverse the default endianness?  */
} sox_encodinginfo_t;

typedef struct sox_oob_t {
  sox_comments_t   comments;         /* comment strings as id=value */
  sox_instrinfo_t  instr;               /* instrument specification */
  sox_loopinfo_t   loops[SOX_MAX_NLOOPS];  /* looping specification */
} sox_oob_t;

typedef char **sox_comments_t;

typedef sox_int32_t sox_sample_t;   /* native SoX audio sample type */

typedef enum sox_option_t {
  sox_option_no,                    /* option specified as no  = 0 */
  sox_option_yes,                   /* option specified as yes = 1 */
  sox_option_default                /* option unspecified      = 2 */
} sox_option_t;

typedef enum sox_bool {
  sox_false,                                          /* false = 0 */
  sox_true                                            /* true  = 1 */
} sox_bool;

typedef enum sox_encoding_t {
SOX_ENCODING_UNKNOWN   ,/* encoding has not yet been determined      */
SOX_ENCODING_SIGN2     ,/* signed linear 2's comp: Mac               */
SOX_ENCODING_UNSIGNED  ,/* unsigned linear: Sound Blaster            */
SOX_ENCODING_FLOAT     ,/* floating point (binary format)            */
SOX_ENCODING_FLOAT_TEXT,/* floating point (text format)              */
SOX_ENCODING_FLAC      ,/* FLAC compression                          */
SOX_ENCODING_HCOM      ,/* Mac FSSD files with Huffman compression   */
SOX_ENCODING_WAVPACK   ,/* WavPack with integer samples              */
SOX_ENCODING_WAVPACKF  ,/* WavPack with float samples                */
SOX_ENCODING_ULAW      ,/* u-law signed logs: US telephony/SPARC     */
SOX_ENCODING_ALAW      ,/* A-law signed logs: non-US telephony/Psion */
SOX_ENCODING_G721      ,/* G.721 4-bit ADPCM                         */
SOX_ENCODING_G723      ,/* G.723 3 or 5 bit ADPCM                    */
SOX_ENCODING_CL_ADPCM  ,/* Creative Labs 8->2,3,4-bit compressed PCM */
SOX_ENCODING_CL_ADPCM16,/* Creative Labs 16->4-bit compressed PCM    */
SOX_ENCODING_MS_ADPCM  ,/* Microsoft compressed PCM                  */
```

```
SOX_ENCODING_IMA_ADPCM ,/* IMA compressed PCM                          */
SOX_ENCODING_OKI_ADPCM ,/* Dialogic/OKI compressed PCM                 */
SOX_ENCODING_DPCM      ,/* Differential PCM: Fasttracker 2 (xi)        */
SOX_ENCODING_DWVW      ,/* Delta Width Variable Word                   */
SOX_ENCODING_DWVWN     ,/* Delta Width Variable Word N-bit             */
SOX_ENCODING_GSM       ,/* GSM 6.10 33byte frame lossy compression     */
SOX_ENCODING_MP3       ,/* MP3 compression                             */
SOX_ENCODING_VORBIS    ,/* Vorbis compression                          */
SOX_ENCODING_AMR_WB    ,/* AMR-WB compression                          */
SOX_ENCODING_AMR_NB    ,/* AMR-NB compression                          */
SOX_ENCODING_CVSD      ,/* Continuously Variable Slope Delta           */
SOX_ENCODING_LPC10     ,/* Linear Predictive Coding                    */
SOX_ENCODING_OPUS      ,/* Opus compression                            */
SOX_ENCODING_DSD       ,/* Direct Stream Digital                       */
SOX_ENCODINGS           /* end of list marker                          */
} sox_encoding_t;
```

DESCRIPTION

libsox_ng is a library of sound sample file format readers/writers and sound effects processors. It is mainly developed to be used by SoX but any audio application might find it useful.

This is not really a manual for the libsox_ng API; it is an introduction to the basic operations for reading and writing audio files and how to use effect chains, followed by notes on how to write new format handlers and effects. For an exhaustive description of the libsox_ng API see http://martinwguy.net/test/soxygen/sox_8h.html

int sox_format_init(void)

sox_format_init() performs initialization required by all file format handlers. If compiled with dynamic library support, this will detect and initialize all external libraries. It should be called before any file operations are performed.

void sox_format_quit(void)

sox_format_quit() performs cleanup related to all file format handlers and should be called after all file operations are completed.

sox_format_t *sox_open_read(char const *_path_, sox_signalinfo_t const *_signal_, sox_encodinginfo_t const *_encoding_, char const *_filetype_)

sox_open_read() opens a file for reading. A special name of '**-**' reads data from stdin. If *signal* is not NULL, it specifies the properties of the audio signal such as the sample rate or the number of channels. If *encoding* is not NULL it specifies the sample encoding. Both *signal* and *encoding* are normally only needed for headerless audio files where that information is not stored in the file. If *filetype* is not NULL, it specifies the file type as the short strings listed in **soxformat_ng(7)**; otherwise, the file's type is guessed from the filename extension and/or the file's contents.

Upon successful completion, **sox_open_read()** returns a pointer to a filled *sox_format_t*, which should eventually be closed with **sox_close()**, or NULL otherwise. Currently there is no way to determine the reason for failure except for the error message printed on the standard error output (stderr).

sox_format_t *sox_open_write(char const *_filename_, sox_signalinfo_t const *_signal_, sox_encodinginfo_t const *_encoding_, char const *_filetype_, sox_oob_t const *_oob_, sox_bool (*overwrite_permitted)(char const *_filename_))

sox_open_write() opens the named file for writing. A special name of '**-**' writes data to stdout. If *signal* is not NULL, it specifies the data format of the output file; the *signal* structure filled in by **sox_open_read()** can be used to copy data in the same format. If *encoding* is not NULL, it specifies the desired sample encoding. Since most file formats can encode data in different ways, this usually has to be specified; if it is NULL, a default is used. If *filetype* is not NULL, it specifies the type of file to write, using the short strings listed in **soxformat_ng**(7); otherwise, the file type is

that of the filename's extension. If *oob* is not NULL and the file type supports comments and other out-of-band data like loop points and instrument information, these are stored in the file header. If *overwrite* is not NULL and the file already exists, the function is called with the filename as its argument to determine whether the file should be overwritten. If it is NULL, existing files are overwritten.

sox_open_write() returns a pointer to a *sox_format_t* which must be closed with **sox_close()**, or NULL if it fails. Like **sox_open_read()**, there is no way to determine the reason for failure except for the error message it prints.

size_t sox_read(sox_format_t *_format_, sox_sample_t *_buf_, size_t _len_)

sox_read() reads *len* samples into *buf* using the format handler specified by *format*. It is the caller's responsibility to provide a large enough buffer for up to *len* samples. All data is converted to 32-bit signed samples before being placed in *buf*. The value of *len* is the total number of samples (number of sample frames multiplied by the number of channels) and if its value is not a multiple of the number of channels, anything might happen.

It returns the number of samples read, which may be less tha the number requested, or zero at the end of the file or if an error occurred. **sox_read()**'s return value does not distinguish between the end of a file and an error but you can inspect **format->sox_errno** to tell the difference. It should be **SOX_SUCCESS** or **SOX_EOF** unless some other error has occurred previously; you can clear **format->sox_errno** to 0 beforehand to be sure and, if it's something other than **SOX_SUCCESS**, there will be a more explicit reason in **format->sox_errstr**.

size_t sox_read(sox_format_t *_format_, sox_sample_t *_buf_, size_t _len_)

sox_read() returns the number of samples successfully read, If an error occurs or the end of the file has been reached, the return value is zero or **SOX_EOF**, depending on the function.

size_t sox_write(sox_format_t *_format_, sox_sample_t const *_buf_, size_t _len_)

sox_write() writes *len* samples from *buf* to the file described by *format* and the 32-bit signed data in *buf* are converted according to the format. The value of *len* is the total samples and must be divisible by the number of channels, otherwise unexpected things will occur. It returns the actual number of samples encoded, zero if an error occurred.

int sox_seek(sox_format_t *_format_, size_t _offset_, int _whence_)

sox_seek() repositions the offset of the file associated with *format* to the given *offset*. On success, it returns **SOX_SUCCESS**; **SOX_EOF** otherwise. Since **sox_read()** and **sox_write()** carry out complex transformations of files' data, its usefulness is questionable.

int sox_close(sox_format_t *_format_)

sox_close() disassociates a **format** from its underlying file. If the format handler was being used for output, any buffered data are written out first. It returns **SOX_SUCCESS** if all went well; if not, **SOX_EOF** or, occasionally, some other error code. Like **sox_open_read()** and **sox_open_write()**, there is currently no way to determine the precise reason for failure other than the messages printed to stderr. In either case, no further use should be made to the handle, not even another call to **sox_close()**.

EFFECTS

sox_effect_handler_t const *sox_find_effect(char const *_name_)

sox_find_effect() returns a pointer to the named effect's handler if it exists, NULL otherwise.

sox_effect_t *sox_create_effect(sox_effect_handler_t const *_effect_)

sox_create_effect() instantiates an effect into a *sox_effect_t* given its *handler*. Any missing methods are automatically set to the appropriate **nothing** method. It returns a pointer to the new effect or NULL if it was not found or had problems starting up.

int sox_effect_options(sox_effect_t *_effect_, int _argc_, char * const _argv_[])

sox_effect_options() passes options into an effect to control its behavior. If it succeeds, *effect->in_signal* should contain the rate and channel count at which it requires input data and *effect->out_signal* the rate and channel count it outputs. When it is present, this information is used

to ensure that appropriate effects are placed in the effects chain to handle any needed conversions. It returns the number of arguments consumed or **SOX_EOF** if any invalid options were passed.

Setting options is only supported before the effect is started. The behavior is undefined if its called when the effect has already started.

sox_effects_chain_t *sox_create_effects_chain(sox_encodinginfo_t const *_in_enc_, sox_encodinginfo_t const *_out_enc_)

sox_create_effects_chain() creates an effects chain to which effects can be added. *in_enc* and *out_enc* are the signal encoding of the chain's input and output. The pointers to *in_enc* and *out_enc* are stored internally, so their memory should not be freed until the effect chain has been deleted. Their values may change to reflect new input or output encodings when effects start up or are restarted. It returns a pointer to the new chain or NULL if something went wrong.

int sox_add_effect(sox_effects_chain_t *_chain_, sox_effect_t *_effect_, sox_signalinfo_t *_in_, sox_signalinfo_t const *_out_)

sox_add_effect adds an effect to a chain. *in* specifies the input signal info for the effect, *out* is a suggestion as to what the output signal should be but, depending on the effect options given and on *in*, the effect may choose a different encoding such as changing the number of channels or the sample rate. Whatever output rate and channels the effect produces are written back to *in* ready to be passed to subsequent calls of **sox_add_effect** so that changes propagate to each new effect. It returns SOX_SUCCESS if it was successful.

void sox_delete_effects_chain(sox_effects_chain_t *_ecp_)

sox_delete_effects_chain() closes an effects chain down and releases any resources reserved during the creation of the chain. It also deletes all the effects in the chain.

EXAMPLES

SoX includes skeleton C files to assist you in writing new formats (**skelform.c**) and effects (**skeleff.c**). 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_ng** and **sox_ng.c** itself is also a good reference.

INTERNALS

SoX's formats and effects operate with an internal sample format of signed 32-bit integers. 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 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 audio. For example, if an effect to mix down left and right channels into one monophonic channel were to use the obvious

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

distortion may occur since the intermediate addition can overflow 32 bits.

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

would get round the overflow but at the expense of the least significant bit.

Stereo data is stored with the left and right speaker data in successive samples and quadraphonic data is stored 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 stored 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 audio: sampling rate, data size (8, 16 or 32 bits), encoding (unsigned, signed, floating point etc.), and the 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 format's private structure.

The *private* area is 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 an example of 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 into the header. The private area is usually not very large and some effects, such as **echo**, **lsx_malloc()** larger areas for delay lines and such.

A *format* has 6 routines:

startread	Set up the format parameters, read in a data header and do anything else that 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. It returns the number of samples actually read.
stopread	Do what needs to be done when it has finished reading.
startwrite	Set up the format parameters, maybe write out a data header and any other preliminaries for writing the format.
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, it will return a lesser number of samples.
stopwrite	Typically, fix up the file header or whatever else 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 according to the following flow diagram:

```
LOOP (invocations with different parameters)
  getopts
  LOOP (invocations with the same parameters)
    LOOP (channels)
      start
    LOOP (while there is input audio to process)
      LOOP (channels)
        flow
    LOOP (while there is output audio to generate)
      LOOP (channels)
        drain
    LOOP (channels)
      stop
  kill
```

Functions that an effect does not need can be NULL. An effect that is marked 'MCHAN' does not use the LOOP (channels) lines and must perform multiple channel processing inside the affected functions. Multiple effect instances may be processed 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**, this means that the effect will not read any more data and can be used to switch to drain mode sooner.
drain	is called when there are no more input data samples. If the effect wishes to generate more data samples, it copies the generated data into the given buffer and returns the number of samples generated. If it fills the buffer, it will be called again;

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 close or free resources such as memory and temporary files that were allocated during *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

How you link against libsox_ng depends on how SoX was built on your system. For a static build, just link against the library. For a dynamic build, use **libtool** to link with the correct linker flags. See the **libtool** manual for details; basically, you use it like this:

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

COPYRIGHT

Copyright 1991−2015 Lance Norskog, Chris Bagwell and sundry contributors.

This library is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

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 General Public License for more details.

AUTHORS

The many authors and contributors are listed in the README file that is distributed with the source code.

SEE ALSO

sox_ng(1), **soxformat_ng**(7), **src/example*.c** in the SoX source distribution.