# **libsndfile**

Libsndfile is a library designed to allow the reading and writing of many different sampled sound file formats (such as MS Windows WAV and the Apple/SGI AIFF format) through one standard library interface.

During read and write operations, formats are seamlessly converted between the format the application program has requested or supplied and the file's data format. The application programmer can remain blissfully unaware of issues such as file endian-ness and data format. See [Note 1](#35nkun2) and [Note 2](#1ksv4uv).

Every effort is made to keep these documents up-to-date, error free and unambiguous. However, since maintaining the documentation is the least fun part of working on libsndfile, these docs can and do fall behind the behaviour of library. If any errors, omissions or ambiguities are found, please notify me (erikd) at mega-nerd dot com.

To supplement this reference documentation, there are simple example programs included in the source code tarball. The test suite which is also part of the source code tarball is also a good place to look for the correct usage of the library functions.

**Finally, if you think there is some feature missing from libsndfile, check that it isn't already implemented (and documented)** [**here**](http://docs.google.com/command.html)**.**

## **Synopsis**

The functions of libsndfile are defined as follows:

#include <stdio.h>  
 #include <sndfile.h>  
  
 SNDFILE\* [sf\_open](#gjdgxs) (const char \*path, int mode, SF\_INFO \*sfinfo) ;  
 SNDFILE\* [sf\_open\_fd](#30j0zll) (int fd, int mode, SF\_INFO \*sfinfo, int close\_desc) ;  
 SNDFILE\* [sf\_open\_virtual](#1fob9te) (SF\_VIRTUAL\_IO \*sfvirtual, int mode, SF\_INFO \*sfinfo, void \*user\_data) ;  
 int [sf\_format\_check](#3znysh7) (const SF\_INFO \*info) ;  
  
 sf\_count\_t [sf\_seek](#2et92p0) (SNDFILE \*sndfile, sf\_count\_t frames, int whence) ;  
  
 int [sf\_command](http://docs.google.com/command.html) (SNDFILE \*sndfile, int cmd, void \*data, int datasize) ;  
  
 int [sf\_error](#tyjcwt) (SNDFILE \*sndfile) ;  
 const char\* [sf\_strerror](#tyjcwt) (SNDFILE \*sndfile) ;  
 const char\* [sf\_error\_number](#tyjcwt) (int errnum) ;  
  
 int [sf\_perror](#tyjcwt) (SNDFILE \*sndfile) ;  
 int [sf\_error\_str](#tyjcwt) (SNDFILE \*sndfile, char\* str, size\_t len) ;  
  
 int [sf\_close](#3dy6vkm) (SNDFILE \*sndfile) ;  
 void [sf\_write\_sync](#1t3h5sf) (SNDFILE \*sndfile) ;  
  
 sf\_count\_t [sf\_read\_short](#4d34og8) (SNDFILE \*sndfile, short \*ptr, sf\_count\_t items) ;  
 sf\_count\_t [sf\_read\_int](#4d34og8) (SNDFILE \*sndfile, int \*ptr, sf\_count\_t items) ;  
 sf\_count\_t [sf\_read\_float](#4d34og8) (SNDFILE \*sndfile, float \*ptr, sf\_count\_t items) ;  
 sf\_count\_t [sf\_read\_double](#4d34og8) (SNDFILE \*sndfile, double \*ptr, sf\_count\_t items) ;  
  
 sf\_count\_t [sf\_readf\_short](#2s8eyo1) (SNDFILE \*sndfile, short \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t [sf\_readf\_int](#2s8eyo1) (SNDFILE \*sndfile, int \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t [sf\_readf\_float](#2s8eyo1) (SNDFILE \*sndfile, float \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t [sf\_readf\_double](#2s8eyo1) (SNDFILE \*sndfile, double \*ptr, sf\_count\_t frames) ;  
  
 sf\_count\_t [sf\_write\_short](#17dp8vu) (SNDFILE \*sndfile, short \*ptr, sf\_count\_t items) ;  
 sf\_count\_t [sf\_write\_int](#17dp8vu) (SNDFILE \*sndfile, int \*ptr, sf\_count\_t items) ;  
 sf\_count\_t [sf\_write\_float](#17dp8vu) (SNDFILE \*sndfile, float \*ptr, sf\_count\_t items) ;  
 sf\_count\_t [sf\_write\_double](#17dp8vu) (SNDFILE \*sndfile, double \*ptr, sf\_count\_t items) ;  
  
 sf\_count\_t [sf\_writef\_short](#3rdcrjn) (SNDFILE \*sndfile, short \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t [sf\_writef\_int](#3rdcrjn) (SNDFILE \*sndfile, int \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t [sf\_writef\_float](#3rdcrjn) (SNDFILE \*sndfile, float \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t [sf\_writef\_double](#3rdcrjn) (SNDFILE \*sndfile, double \*ptr, sf\_count\_t frames) ;  
  
 sf\_count\_t [sf\_read\_raw](#26in1rg) (SNDFILE \*sndfile, void \*ptr, sf\_count\_t bytes) ;  
 sf\_count\_t [sf\_write\_raw](#26in1rg) (SNDFILE \*sndfile, void \*ptr, sf\_count\_t bytes) ;  
  
 const char\* [sf\_get\_string](#lnxbz9) (SNDFILE \*sndfile, int str\_type) ;  
 int [sf\_set\_string](#lnxbz9) (SNDFILE \*sndfile, int str\_type, const char\* str) ;

SNDFILE\* is an anonymous pointer to data which is private to the library.

## **File Open Function**

SNDFILE\* sf\_open (const char \*path, int mode, SF\_INFO \*sfinfo) ;

The SF\_INFO structure is for passing data between the calling function and the library when opening a file for reading or writing. It is defined in sndfile.h as follows:

typedef struct  
 { sf\_count\_t frames ; /\* Used to be called samples. \*/  
 int samplerate ;  
 int channels ;  
 int format ;  
 int sections ;  
 int seekable ;  
 } SF\_INFO ;

The mode parameter for this function can be any one of the following three values:

SFM\_READ - read only mode  
 SFM\_WRITE - write only mode  
 SFM\_RDWR - read/write mode

When opening a file for read, the **format** field should be set to zero before calling sf\_open(). The only exception to this is the case of RAW files where the caller has to set the samplerate, channels and format fields to valid values. All other fields of the structure are filled in by the library.

When opening a file for write, the caller must fill in structure members samplerate, channels, and format.

The format field in the above SF\_INFO structure is made up of the bit-wise OR of a major format type (values between 0x10000 and 0x08000000), a minor format type (with values less than 0x10000) and an optional endian-ness value. The currently understood formats are listed in sndfile.h as follows and also include bitmasks for separating major and minor file types. Not all combinations of endian-ness and major and minor file types are valid.

enum  
 { /\* Major formats. \*/  
 SF\_FORMAT\_WAV = 0x010000, /\* Microsoft WAV format (little endian). \*/  
 SF\_FORMAT\_AIFF = 0x020000, /\* Apple/SGI AIFF format (big endian). \*/  
 SF\_FORMAT\_AU = 0x030000, /\* Sun/NeXT AU format (big endian). \*/  
 SF\_FORMAT\_RAW = 0x040000, /\* RAW PCM data. \*/  
 SF\_FORMAT\_PAF = 0x050000, /\* Ensoniq PARIS file format. \*/  
 SF\_FORMAT\_SVX = 0x060000, /\* Amiga IFF / SVX8 / SV16 format. \*/  
 SF\_FORMAT\_NIST = 0x070000, /\* Sphere NIST format. \*/  
 SF\_FORMAT\_VOC = 0x080000, /\* VOC files. \*/  
 SF\_FORMAT\_IRCAM = 0x0A0000, /\* Berkeley/IRCAM/CARL \*/  
 SF\_FORMAT\_W64 = 0x0B0000, /\* Sonic Foundry's 64 bit RIFF/WAV \*/  
 SF\_FORMAT\_MAT4 = 0x0C0000, /\* Matlab (tm) V4.2 / GNU Octave 2.0 \*/  
 SF\_FORMAT\_MAT5 = 0x0D0000, /\* Matlab (tm) V5.0 / GNU Octave 2.1 \*/  
 SF\_FORMAT\_PVF = 0x0E0000, /\* Portable Voice Format \*/  
 SF\_FORMAT\_XI = 0x0F0000, /\* Fasttracker 2 Extended Instrument \*/  
 SF\_FORMAT\_HTK = 0x100000, /\* HMM Tool Kit format \*/  
 SF\_FORMAT\_SDS = 0x110000, /\* Midi Sample Dump Standard \*/  
 SF\_FORMAT\_AVR = 0x120000, /\* Audio Visual Research \*/  
 SF\_FORMAT\_WAVEX = 0x130000, /\* MS WAVE with WAVEFORMATEX \*/  
 SF\_FORMAT\_SD2 = 0x160000, /\* Sound Designer 2 \*/  
 SF\_FORMAT\_FLAC = 0x170000, /\* FLAC lossless file format \*/  
 SF\_FORMAT\_CAF = 0x180000, /\* Core Audio File format \*/  
 SF\_FORMAT\_WVE = 0x190000, /\* Psion WVE format \*/  
 SF\_FORMAT\_OGG = 0x200000, /\* Xiph OGG container \*/  
 SF\_FORMAT\_MPC2K = 0x210000, /\* Akai MPC 2000 sampler \*/  
 SF\_FORMAT\_RF64 = 0x220000, /\* RF64 WAV file \*/  
  
 /\* Subtypes from here on. \*/  
  
 SF\_FORMAT\_PCM\_S8 = 0x0001, /\* Signed 8 bit data \*/  
 SF\_FORMAT\_PCM\_16 = 0x0002, /\* Signed 16 bit data \*/  
 SF\_FORMAT\_PCM\_24 = 0x0003, /\* Signed 24 bit data \*/  
 SF\_FORMAT\_PCM\_32 = 0x0004, /\* Signed 32 bit data \*/  
  
 SF\_FORMAT\_PCM\_U8 = 0x0005, /\* Unsigned 8 bit data (WAV and RAW only) \*/  
  
 SF\_FORMAT\_FLOAT = 0x0006, /\* 32 bit float data \*/  
 SF\_FORMAT\_DOUBLE = 0x0007, /\* 64 bit float data \*/  
  
 SF\_FORMAT\_ULAW = 0x0010, /\* U-Law encoded. \*/  
 SF\_FORMAT\_ALAW = 0x0011, /\* A-Law encoded. \*/  
 SF\_FORMAT\_IMA\_ADPCM = 0x0012, /\* IMA ADPCM. \*/  
 SF\_FORMAT\_MS\_ADPCM = 0x0013, /\* Microsoft ADPCM. \*/  
  
 SF\_FORMAT\_GSM610 = 0x0020, /\* GSM 6.10 encoding. \*/  
 SF\_FORMAT\_VOX\_ADPCM = 0x0021, /\* Oki Dialogic ADPCM encoding. \*/  
  
 SF\_FORMAT\_G721\_32 = 0x0030, /\* 32kbs G721 ADPCM encoding. \*/  
 SF\_FORMAT\_G723\_24 = 0x0031, /\* 24kbs G723 ADPCM encoding. \*/  
 SF\_FORMAT\_G723\_40 = 0x0032, /\* 40kbs G723 ADPCM encoding. \*/  
  
 SF\_FORMAT\_DWVW\_12 = 0x0040, /\* 12 bit Delta Width Variable Word encoding. \*/  
 SF\_FORMAT\_DWVW\_16 = 0x0041, /\* 16 bit Delta Width Variable Word encoding. \*/  
 SF\_FORMAT\_DWVW\_24 = 0x0042, /\* 24 bit Delta Width Variable Word encoding. \*/  
 SF\_FORMAT\_DWVW\_N = 0x0043, /\* N bit Delta Width Variable Word encoding. \*/  
  
 SF\_FORMAT\_DPCM\_8 = 0x0050, /\* 8 bit differential PCM (XI only) \*/  
 SF\_FORMAT\_DPCM\_16 = 0x0051, /\* 16 bit differential PCM (XI only) \*/  
  
 SF\_FORMAT\_VORBIS = 0x0060, /\* Xiph Vorbis encoding. \*/  
  
 /\* Endian-ness options. \*/  
  
 SF\_ENDIAN\_FILE = 0x00000000, /\* Default file endian-ness. \*/  
 SF\_ENDIAN\_LITTLE = 0x10000000, /\* Force little endian-ness. \*/  
 SF\_ENDIAN\_BIG = 0x20000000, /\* Force big endian-ness. \*/  
 SF\_ENDIAN\_CPU = 0x30000000, /\* Force CPU endian-ness. \*/  
  
 SF\_FORMAT\_SUBMASK = 0x0000FFFF,  
 SF\_FORMAT\_TYPEMASK = 0x0FFF0000,  
 SF\_FORMAT\_ENDMASK = 0x30000000  
 } ;

Every call to sf\_open() should be matched with a call to sf\_close() to free up memory allocated during the call to sf\_open().

On success, the sf\_open function returns a non-NULL pointer which should be passed as the first parameter to all subsequent libsndfile calls dealing with that audio file. On fail, the sf\_open function returns a NULL pointer. An explanation of the error can obtained by passing NULL to [sf\_strerror](#tyjcwt).

### **File Descriptor Open**

SNDFILE\* sf\_open\_fd (int fd, int mode, SF\_INFO \*sfinfo, int close\_desc) ;

**Note:** On Microsoft Windows, this function does not work if the application and the libsndfile DLL are linked to different versions of the Microsoft C runtime DLL.

The second open function takes a file descriptor of a file that has already been opened. Care should be taken to ensure that the mode of the file represented by the descriptor matches the mode argument. This function is useful in the following circumstances:

* Opening temporary files securely (ie use the tmpfile() to return a FILE\* pointer and then using fileno() to retrieve the file descriptor which is then passed to libsndfile).
* Opening files with file names using OS specific character encodings and then passing the file descriptor to sf\_open\_fd().
* Opening sound files embedded within larger files. [More info](http://docs.google.com/embedded_files.html).

Every call to sf\_open\_fd() should be matched with a call to sf\_close() to free up memory allocated during the call to sf\_open().

When sf\_close() is called, the file descriptor is only closed if the **close\_desc** parameter was TRUE when the sf\_open\_fd() function was called.

On success, the sf\_open\_fd function returns a non-NULL pointer which should be passed as the first parameter to all subsequent libsndfile calls dealing with that audio file. On fail, the sf\_open\_fd function returns a NULL pointer.

### **Virtual File Open Function**

SNDFILE\* sf\_open\_virtual (SF\_VIRTUAL\_IO \*sfvirtual, int mode, SF\_INFO \*sfinfo, void \*user\_data) ;

Opens a soundfile from a virtual file I/O context which is provided by the caller. This is usually used to interface libsndfile to a stream or buffer based system. Apart from the sfvirtual and the user\_data parameters this function behaves like [sf\_open](#gjdgxs).

typedef struct  
 { sf\_vio\_get\_filelen get\_filelen ;  
 sf\_vio\_seek seek ;  
 sf\_vio\_read read ;  
 sf\_vio\_write write ;  
 sf\_vio\_tell tell ;  
 } SF\_VIRTUAL\_IO ;

Libsndfile calls the callbacks provided by the SF\_VIRTUAL\_IO structure when opening, reading and writing to the virtual file context. The user\_data pointer is a user defined context which will be available in the callbacks.

typedef sf\_count\_t (\*sf\_vio\_get\_filelen) (void \*user\_data) ;  
 typedef sf\_count\_t (\*sf\_vio\_seek) (sf\_count\_t offset, int whence, void \*user\_data) ;  
 typedef sf\_count\_t (\*sf\_vio\_read) (void \*ptr, sf\_count\_t count, void \*user\_data) ;  
 typedef sf\_count\_t (\*sf\_vio\_write) (const void \*ptr, sf\_count\_t count, void \*user\_data) ;  
 typedef sf\_count\_t (\*sf\_vio\_tell) (void \*user\_data) ;

#### sf\_vio\_get\_filelen

typedef sf\_count\_t (\*sf\_vio\_get\_filelen) (void \*user\_data) ;

The virtual file contex must return the length of the virtual file in bytes.

#### sf\_vio\_seek

typedef sf\_count\_t (\*sf\_vio\_seek) (sf\_count\_t offset, int whence, void \*user\_data) ;

The virtual file context must seek to offset using the seek mode provided by whence which is one of

SEEK\_CUR  
 SEEK\_SET  
 SEEK\_END

The return value must contain the new offset in the file.

#### sf\_vio\_read

typedef sf\_count\_t (\*sf\_vio\_read) (void \*ptr, sf\_count\_t count, void \*user\_data) ;

The virtual file context must copy ("read") "count" bytes into the buffer provided by ptr and return the count of actually copied bytes.

#### sf\_vio\_write

typedef sf\_count\_t (\*sf\_vio\_write) (const void \*ptr, sf\_count\_t count, void \*user\_data) ;

The virtual file context must process "count" bytes stored in the buffer passed with ptr and return the count of actually processed bytes.

#### sf\_vio\_tell

typedef sf\_count\_t (\*sf\_vio\_tell) (void \*user\_data) ;

Return the current position of the virtual file context.

## **Format Check Function**

int sf\_format\_check (const SF\_INFO \*info) ;

This function allows the caller to check if a set of parameters in the SF\_INFO struct is valid before calling sf\_open (SFM\_WRITE).

sf\_format\_check returns TRUE if the parameters are valid and FALSE otherwise.

## **File Seek Functions**

sf\_count\_t sf\_seek (SNDFILE \*sndfile, sf\_count\_t frames, int whence) ;

The file seek functions work much like lseek in unistd.h with the exception that the non-audio data is ignored and the seek only moves within the audio data section of the file. In addition, seeks are defined in number of (multichannel) frames. Therefore, a seek in a stereo file from the current position forward with an offset of 1 would skip forward by one sample of both channels.

like lseek(), the whence parameter can be any one of the following three values:

SEEK\_SET - The offset is set to the start of the audio data plus offset (multichannel) frames.  
 SEEK\_CUR - The offset is set to its current location plus offset (multichannel) frames.  
 SEEK\_END - The offset is set to the end of the data plus offset (multichannel) frames.

Internally, libsndfile keeps track of the read and write locations using separate read and write pointers. If a file has been opened with a mode of SFM\_RDWR, bitwise OR-ing the standard whence values above with either SFM\_READ or SFM\_WRITE allows the read and write pointers to be modified separately. If the SEEK\_\* values are used on their own, the read and write pointers are both modified.

Note that the frames offset can be negative and in fact should be when SEEK\_END is used for the whence parameter.

sf\_seek will return the offset in (multichannel) frames from the start of the audio data or -1 if an error occured (ie an attempt is made to seek beyond the start or end of the file).

**Error Reporting Functions**

int sf\_error (SNDFILE \*sndfile) ;

This function returns the current error number for the given SNDFILE. The error number may be one of the following:

enum  
 { SF\_ERR\_NO\_ERROR = 0,  
 SF\_ERR\_UNRECOGNISED\_FORMAT = 1,  
 SF\_ERR\_SYSTEM = 2,  
 SF\_ERR\_MALFORMED\_FILE = 3,  
 SF\_ERR\_UNSUPPORTED\_ENCODING = 4  
 } ;

or any one of many other internal error values. Applications should only test the return value against error values defined in <sndfile.h> as the internal error values are subject to change at any time. For errors not in the above list, the function sf\_error\_number() can be used to convert it to an error string.

const char\* sf\_strerror (SNDFILE \*sndfile) ;  
 const char\* sf\_error\_number (int errnum) ;

The error functions sf\_strerror() and sf\_error\_number() convert the library's internal error enumerations into text strings.

int sf\_perror (SNDFILE \*sndfile) ;  
 int sf\_error\_str (SNDFILE \*sndfile, char\* str, size\_t len) ;

The functions sf\_perror() and sf\_error\_str() are deprecated and will be dropped from the library at some later date.

**File Close Function**

int sf\_close (SNDFILE \*sndfile) ;

The close function closes the file, deallocates its internal buffers and returns 0 on success or an error value otherwise.

**Write Sync Function**

void sf\_write\_sync (SNDFILE \*sndfile) ;

If the file is opened SFM\_WRITE or SFM\_RDWR, call the operating system's function to force the writing of all file cache buffers to disk. If the file is opened SFM\_READ no action is taken.

**File Read Functions (Items)**

sf\_count\_t sf\_read\_short (SNDFILE \*sndfile, short \*ptr, sf\_count\_t items) ;  
 sf\_count\_t sf\_read\_int (SNDFILE \*sndfile, int \*ptr, sf\_count\_t items) ;  
 sf\_count\_t sf\_read\_float (SNDFILE \*sndfile, float \*ptr, sf\_count\_t items) ;  
 sf\_count\_t sf\_read\_double (SNDFILE \*sndfile, double \*ptr, sf\_count\_t items) ;

The file read items functions fill the array pointed to by ptr with the requested number of items. The items parameter must be an integer product of the number of channels or an error will occur.

It is important to note that the data type used by the calling program and the data format of the file do not need to be the same. For instance, it is possible to open a 16 bit PCM encoded WAV file and read the data using sf\_read\_float(). The library seamlessly converts between the two formats on-the-fly. See [Note 1](#35nkun2).

The sf\_read\_XXXX functions return the number of items read. Unless the end of the file was reached during the read, the return value should equal the number of items requested. Attempts to read beyond the end of the file will not result in an error but will cause the sf\_read\_XXXX functions to return less than the number of items requested or 0 if already at the end of the file.

**File Read Functions (Frames)**

sf\_count\_t sf\_readf\_short (SNDFILE \*sndfile, short \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t sf\_readf\_int (SNDFILE \*sndfile, int \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t sf\_readf\_float (SNDFILE \*sndfile, float \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t sf\_readf\_double (SNDFILE \*sndfile, double \*ptr, sf\_count\_t frames) ;

The file read frames functions fill the array pointed to by ptr with the requested number of frames of data. The array must be large enough to hold the product of frames and the number of channels.

**Care must be taken to ensure that there is enough space in the array pointed to by ptr, to take (frames \* channels) number of items (shorts, ints, floats or doubles).**

The sf\_readf\_XXXX functions return the number of frames read. Unless the end of the file was reached during the read, the return value should equal the number of frames requested. Attempts to read beyond the end of the file will not result in an error but will cause the sf\_readf\_XXXX functions to return less than the number of frames requested or 0 if already at the end of the file.

**File Write Functions (Items)**

sf\_count\_t sf\_write\_short (SNDFILE \*sndfile, short \*ptr, sf\_count\_t items) ;  
 sf\_count\_t sf\_write\_int (SNDFILE \*sndfile, int \*ptr, sf\_count\_t items) ;  
 sf\_count\_t sf\_write\_float (SNDFILE \*sndfile, float \*ptr, sf\_count\_t items) ;  
 sf\_count\_t sf\_write\_double (SNDFILE \*sndfile, double \*ptr, sf\_count\_t items) ;

The file write items functions write the data in the array pointed to by ptr to the file. The items parameter must be an integer product of the number of channels or an error will occur.

It is important to note that the data type used by the calling program and the data format of the file do not need to be the same. For instance, it is possible to open a 16 bit PCM encoded WAV file and write the data using sf\_write\_float(). The library seamlessly converts between the two formats on-the-fly. See [Note 1](#35nkun2).

The sf\_write\_XXXX functions return the number of items written (which should be the same as the items parameter).

**File Write Functions (Frames)**

sf\_count\_t sf\_writef\_short (SNDFILE \*sndfile, short \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t sf\_writef\_int (SNDFILE \*sndfile, int \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t sf\_writef\_float (SNDFILE \*sndfile, float \*ptr, sf\_count\_t frames) ;  
 sf\_count\_t sf\_writef\_double (SNDFILE \*sndfile, double \*ptr, sf\_count\_t frames) ;

The file write frames functions write the data in the array pointed to by ptr to the file. The array must be large enough to hold the product of frames and the number of channels.

The sf\_writef\_XXXX functions return the number of frames written (which should be the same as the frames parameter).

**Raw File Read and Write Functions**

sf\_count\_t sf\_read\_raw (SNDFILE \*sndfile, void \*ptr, sf\_count\_t bytes) ;  
 sf\_count\_t sf\_write\_raw (SNDFILE \*sndfile, void \*ptr, sf\_count\_t bytes) ;

**Note:** Unless you are writing an external decoder/encode that uses libsndfile to handle the file headers, you should not be using these functions.

The raw read and write functions read raw audio data from the audio file (not to be confused with reading RAW header-less PCM files). The number of bytes read or written must always be an integer multiple of the number of channels multiplied by the number of bytes required to represent one sample from one channel.

The raw read and write functions return the number of bytes read or written (which should be the same as the bytes parameter).

**Note : The result of using of both regular reads/writes and raw reads/writes on compressed file formats other than SF\_FORMAT\_ALAW and SF\_FORMAT\_ULAW is undefined.**

See also : [SFC\_RAW\_NEEDS\_ENDSWAP](http://docs.google.com/command.html#SFC_RAW_NEEDS_ENDSWAP)

**Functions for Reading and Writing String Data**

const char\* sf\_get\_string (SNDFILE \*sndfile, int str\_type) ;  
 int sf\_set\_string (SNDFILE \*sndfile, int str\_type, const char\* str) ;

These functions allow strings to be set on files opened for write and to be retrieved from files opened for read where supported by the given file type. The **str\_type** parameter can be any one of the following string types:

enum  
 { SF\_STR\_TITLE,  
 SF\_STR\_COPYRIGHT,  
 SF\_STR\_SOFTWARE,  
 SF\_STR\_ARTIST,  
 SF\_STR\_COMMENT,  
 SF\_STR\_DATE  
 } ;

The sf\_get\_string() function returns the specified string if it exists and a NULL pointer otherwise. In addition to the string ids above, SF\_STR\_FIRST (== SF\_STR\_TITLE) and SF\_STR\_LAST (always the same as the highest numbers string id) are also available to allow iteration over all the available string ids.

The sf\_set\_string() function sets the string data. It returns zero on success and non-zero on error. The error code can be converted to a string using sf\_error\_number().

## 

**Note 1**

When converting between integer PCM formats of differing size (ie using sf\_read\_int() to read a 16 bit PCM encoded WAV file) libsndfile obeys one simple rule:

Whenever integer data is moved from one sized container to another sized container, the most significant bit in the source container will become the most significant bit in the destination container.

When converting between integer data and floating point data, different rules apply. The default behaviour when reading floating point data (sf\_read\_float() or sf\_read\_double ()) from a file with integer data is normalisation. Regardless of whether data in the file is 8, 16, 24 or 32 bit wide, the data will be read as floating point data in the range [-1.0, 1.0]. Similarly, data in the range [-1.0, 1.0] will be written to an integer PCM file so that a data value of 1.0 will be the largest allowable integer for the given bit width. This normalisation can be turned on or off using the [sf\_command](http://docs.google.com/command.html) interface.

**Note 2**

Reading a file containg floating point data (allowable with WAV, AIFF, AU and other file formats) using integer read methods (sf\_read\_short() or sf\_read\_int()) can produce unexpected results. For instance the data in the file may have a maximum absolute value < 1.0 which would mean that all sample values read from the file will be zero. In order to read these files correctly using integer read methods, it is recommended that you use the [sf\_command](http://docs.google.com/command.html) interface, a command of [SFC\_SET\_SCALE\_FLOAT\_INT\_READ](http://docs.google.com/command.html#SFC_SET_SCALE_FLOAT_INT_READ) and a parameter of SF\_TRUE to force correct scaling.

The libsndfile home page is [here](http://www.mega-nerd.com/libsndfile/).

Version : 1.0.24