SDL_mixer

29 July 2005

Jonathan Atkins

Copyright © 2002 Jonathan Atkins

Permission is granted to distribute freely, or in a distribution of any kind. All distributions of this file must be in an unaltered state, except for corrections.

The latest copy of this document can be found at `http://jcatki.no-ip.org/SDL_mixer`

Table of Contents

1 Overview

A Little Bit About Me

I am currently, as I write this document, a programmer for Raytheon. There I do all sorts of communications, network, GUI, and other general programming tasks in C/C++ on the Solaris and sometimes Linux Operating Systems. I have been programming sound code in my free time for only a little while now. Sound is an integral part to any game. The human senses are mostly starved during video game play. there's only some tactile feedback on some controlers, and of course the eyes are in use but only for about 30% of their viewing area. So to add more we do need sound to help the game player feel more in the action, and to set certain moods as the game progresses. Sound ends up accounting for perhaps 50% or more of a gamers experience. Music and sound effects are all integral parts of the gaming experience. While this document doesn't explain how to get music and samples to use, it will explain how to use them with SDL_mixer.

Feel free to contact me: `jcatki@jcatki.no-ip.org`

I am also usually on IRC at irc.freenode.net in the #SDL channel as LIM

This is the README in the SDL_mixer source archive.

SDL_mixer 1.2

The latest version of this library is available from:
SDL_mixer Homepage (`http://www.libsdl.org/projects/SDL_mixer/`)

Due to popular demand, here is a simple multi-channel audio mixer. It supports 8 channels of 16 bit stereo audio, plus a single channel of music, mixed by the popular MikMod MOD, Timidity MIDI and SMPEG MP3 libraries.

See the header file SDL_mixer.h and the examples playwave.c and playmus.c for documentation on this mixer library.

The mixer can currently load Microsoft WAVE files and Creative Labs VOC files as audio samples, and can load MIDI files via Timidity and the following music formats via MikMod: .MOD .S3M .IT .XM. It can load Ogg Vorbis streams as music if built with the Ogg Vorbis libraries, and finally it can load MP3 music using the SMPEG library.

The process of mixing MIDI files to wave output is very CPU intensive, so if playing regular WAVE files sound great, but playing MIDI files sound choppy, try using 8-bit audio, mono audio, or lower frequencies.

To play MIDI files, you'll need to get a complete set of GUS patches from: Timidity GUS Patches (`http://www.libsdl.org/projects/mixer/timidity/timidity.tar.gz`) and unpack them in /usr/local/lib under UNIX, and C:\ under Win32.

This library is available under the GNU Library General Public License, see the file "COPYING" for details.

2 Getting Started

This assumes you have gotten SDL_mixer and installed it on your system. SDL_mixer has an INSTALL document in the source distribution to help you get it compiled and installed.
Generally, installation consists of:

```
./configure
make
make install
```

SDL_mixer supports playing music and sound samples from the following formats:
- WAVE/RIFF (.wav)
- AIFF (.aiff)
- VOC (.voc)
- MOD (.mod .xm .s3m .669 .it .med and more) using included mikmod
- MIDI (.mid) using timidity or native midi hardware
- OggVorbis (.ogg) requiring ogg/vorbis libraries on system
- MP3 (.mp3) requiring SMPEG library on system
- also any command-line player, which is not mixed by SDL_mixer...

You may also want to look at some demonstration code which may be downloaded from:
`http://jcatki.no-ip.org/SDL_mixer/`

2.1 Includes

To use SDL_mixer functions in a C/C++ source code file, you must use the SDL_mixer.h include file:

```
#include "SDL_mixer.h"
```

2.2 Compiling

To link with SDL_mixer you should use sdl-config to get the required SDL compilation options. After that, compiling with SDL_mixer is quite easy.
Note: Some systems may not have the SDL_mixer library and include file in the same place as the SDL library and includes are located, in that case you will need to add more -I and -L paths to these command lines.

Simple Example for compiling an object file:

```
cc -c `sdl-config --cflags` mysource.c
```

Simple Example for compiling an object file:

```
cc -o myprogram mysource.o `sdl-config --libs` -lSDL_mixer
```

Now `myprogram` is ready to run.

3 Conflicts

When using SDL_mixer functions you need to avoid the following functions from SDL:

`SDL_OpenAudio`
: Use `Mix_OpenAudio` instead.

`SDL_CloseAudio`
: Use `Mix_CloseAudio` instead.

`SDL_PauseAudio`
: Use `Mix_Pause(-1)` and `Mix_PauseMusic` instead, to pause.
Use `Mix_Resume(-1)` and `Mix_ResumeMusic` instead, to unpause.

`SDL_LockAudio`
: This is just not needed since SDL_mixer handles this for you.
Using it may cause problems as well.

`SDL_UnlockAudio`
: This is just not needed since SDL_mixer handles this for you.
Using it may cause problems as well.

You may call the following functions freely:

`SDL_AudioDriverName`
: This will still work as usual.

`SDL_GetAudioStatus`
: This will still work, though it will likely return **SDL_AUDIO_PLAYING** even though SDL_mixer is just playing silence.

It is also a BAD idea to call SDL_mixer and SDL audio functions from a callback. Callbacks include Effects functions and other SDL_mixer audio hooks.

4 Functions

These are the functions in the SDL_mixer API.

4.1 General

These functions are useful, as they are the only/best ways to work with SDL_mixer.

4.1.1 Mix_Linked_Version

`const SDL_version *`**Mix_Linked_Version**`()`
`void` **MIX_VERSION**`(SDL_version *compile_version)`

This works similar to `SDL_Linked_Version` and **SDL_VERSION**.
Using these you can compare the runtime version to the version that you compiled with.

```
SDL_version compile_version, *link_version;
MIX_VERSION(&compile_version);
printf("compiled with SDL_mixer version: %d.%d.%d\n",
        compile_version.major,
        compile_version.minor,
        compile_version.patch);
link_version=Mix_Linked_Version();
printf("running with SDL_mixer version: %d.%d.%d\n",
        link_version->major,
        link_version->minor,
        link_version->patch);
```

See Also:
⟨undefined⟩ [Mix_OpenAudio], page ⟨undefined⟩, ⟨undefined⟩ [Mix_QuerySpec], page ⟨undefined⟩

4.1.2 Mix_OpenAudio

`int` **Mix_OpenAudio**`(int` *frequency*`, Uint16` *format*`, int` *channels*`, int` *chunksize*`)`

frequency Output sampling frequency in samples per second (Hz).
you might use **MIX_DEFAULT_FREQUENCY**(22050) since that is a good value for most games.

format Output sample format.

channels Number of sound channels in output.
Set to 2 for stereo, 1 for mono. This has nothing to do with mixing channels.

chunksize Bytes used per output sample.

Initialize the mixer API.
This must be called before using other functions in this library.
SDL must be initialized with **SDL_INIT_AUDIO** before this call. *frequency* would be 44100 for 44.1KHz, which is CD audio rate. Most games use 22050, because 44100 requires too much CPU power on older computers. *chunksize* is the size of each mixed sample. The smaller this is the more your hooks will be called. If make this too small on a slow system, sound may skip. If made to large, sound effects will lag behind the action more. You want a happy medium for your target computer. You also may make this 4096, or larger, if you are just playing music. **MIX_CHANNELS**(8) mixing channels will be allocated by default. You may call this function multiple times, however you will have to call Mix_CloseAudio just as many times for the device to actually close. The format will not changed on subsequent calls. So you will have to close all the way before trying to open with different format parameters.
format is based on SDL audio support, see SDL_audio.h. Here are the values listed there:

AUDIO_U8
Unsigned 8-bit samples

AUDIO_S8
Signed 8-bit samples

AUDIO_U16LSB
Unsigned 16-bit samples, in little-endian byte order

AUDIO_S16LSB
Signed 16-bit samples, in little-endian byte order

AUDIO_U16MSB
Unsigned 16-bit samples, in big-endian byte order

AUDIO_S16MSB
Signed 16-bit samples, in big-endian byte order

AUDIO_U16
same as **AUDIO_U16LSB** (for backwards compatability probably)

AUDIO_S16
same as **AUDIO_S16LSB** (for backwards compatability probably)

AUDIO_U16SYS
Unsigned 16-bit samples, in system byte order

AUDIO_S16SYS
Signed 16-bit samples, in system byte order

MIX_DEFAULT_FORMAT is the same as **AUDIO_S16SYS**.

Returns: 0 on success, -1 on errors

```
// start SDL with audio support
if(SDL_Init(SDL_INIT_AUDIO)==-1) {
    printf("SDL_Init: %s\n", SDL_GetError());
    exit(1);
}
// open 44.1KHz, signed 16bit, system byte order,
//      stereo audio, using 1024 byte chunks
if(Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 1024)==-1) {
    printf("Mix_OpenAudio: %s\n", Mix_GetError());
    exit(2);
}
```

See Also:
⟨undefined⟩ [Mix_CloseAudio], page ⟨undefined⟩, ⟨undefined⟩ [Mix_QuerySpec], page ⟨undefined⟩, ⟨undefined⟩ [Mix_AllocateChannels], page ⟨undefined⟩

4.1.3 Mix_CloseAudio

`void` **Mix_CloseAudio**`()`

Shutdown and cleanup the mixer API.
After calling this all audio is stopped, the device is closed, and the SDL_mixer functions should not be used. You may, of course, use Mix_OpenAudio to start the functionality again.
Note: This function doesn't do anything until you have called it the same number of times that you called Mix_OpenAudio. You may use `Mix_QuerySpec` to find out how many times `Mix_CloseAudio` needs to be called before the device is actually closed.

```
Mix_CloseAudio();
// you could SDL_Quit(); here...or not.
```

See Also:
⟨undefined⟩ [Mix_OpenAudio], page ⟨undefined⟩, ⟨undefined⟩ [Mix_QuerySpec], page ⟨undefined⟩

4.1.4 Mix_SetError

`void` **Mix_SetError**`(const char *`*`fmt`*`, ...)`

This is the same as SDL_SetError, which sets the error string which may be fetched with Mix_GetError (or SDL_GetError). This functions acts like printf, except that it is limited to **SDL_ERRBUFIZE**(1024) chars in length. It only accepts the following format types: `%s, %d, %f, %p`. No variations are supported, like `%.2f` would not work. For any more specifics read the SDL docs.

```
int mymixfunc(int i) {
    Mix_SetError("mymixfunc is not implemented! %d was passed in.",i);
    return(-1);
}
```

See Also:
⟨undefined⟩ [Mix_GetError], page ⟨undefined⟩

4.1.5 Mix_GetError

char *Mix_GetError()

This is the same as SDL_GetError, which returns the last error set as a string which you may use to tell the user what happened when an error status has been returned from an SDL_mixer function call.

Returns: a char pointer (string) containing a humam readble version or the reason for the last error that occured.

```
printf("Oh My Goodness, an error : %s", Mix_GetError());
```

See Also:
⟨undefined⟩ [Mix_SetError], page ⟨undefined⟩

4.1.6 Mix_QuerySpec

`int` **Mix_QuerySpec**`(int *`*frequency*`, Uint16 *`*format*`, int *`*channels*`)`

frequency A pointer to an int where the frequency actually used by the opened audio device will be stored.

format A pointer to a Uint16 where the output format actually being used by the audio device will be stored.

channels A pointer to an int where the number of audio channels will be stored. 2 will mean stereo, 1 will mean mono.

Get the actual audio format in use by the opened audio device. This may or may not match the parameters you passed to **Mix_OpenAudio**.

Returns: 0 on error. If the device was open the number of times it was opened will be returned. The values of the arguments variables are not set on an error.

```
// get and print the audio format in use
int numtimesopened, frequency, channels;
Uint16 format;
numtimesopened=Mix_QuerySpec(&frequency, &format, &channels);
if(!numtimesopened) {
    printf("Mix_QuerySpec: %s\n",Mix_GetError());
}
else {
    char *format_str="Unknown";
    switch(format) {
        case AUDIO_U8: format_str="U8"; break;
        case AUDIO_S8: format_str="S8"; break;
        case AUDIO_U16LSB: format_str="U16LSB"; break;
        case AUDIO_S16LSB: format_str="S16LSB"; break;
        case AUDIO_U16MSB: format_str="U16MSB"; break;
        case AUDIO_S16MSB: format_str="S16MSB"; break;
    }
    printf("opened=%d times  frequency=%dHz  format=%s  channels=%d",
            numtimesopened, frequency, format_str, channels);
}
```

See Also:
⟨undefined⟩ [Mix_OpenAudio], page ⟨undefined⟩

4.2 Samples

These functions work with `Mix_Chunk` samples.

4.2.1 Mix_LoadWAV

`Mix_Chunk *`**Mix_LoadWAV**`(char *`*file*`)`

file File name to load sample from.

Load *file* for use as a sample. This is actually `Mix_LoadWAV_RW(SDL_RWFromFile(`*file*`, "rb"), 1)`. This can load WAVE, AIFF, RIFF, OGG, and VOC files.
Note: You must call SDL_OpenAudio before this. It must know the output characteristics so it can convert the sample for playback, it does this conversion at load time.

Returns: a pointer to the sample as a `Mix_Chunk`. **NULL** is returned on errors.

```
// load sample.wav in to sample
Mix_Chunk *sample;
sample=Mix_LoadWAV("sample.wav");
if(!sample) {
    printf("Mix_LoadWAV: %s\n", Mix_GetError());
    // handle error
}
```

See Also:
⟨undefined⟩ [Mix_LoadWAV_RW], page ⟨undefined⟩, ⟨undefined⟩ [Mix_QuickLoad_WAV], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FreeChunk], page ⟨undefined⟩

4.2.2 Mix_LoadWAV_RW

`Mix_Chunk *`**Mix_LoadWAV_RW**`(SDL_RWops *`*`src`*`, int `*`freesrc`*`)`

src The source SDL_RWops as a pointer. The sample is loaded from this.

freesrc A non-zero value mean is will automatically close/free the *src* for you.

Load *src* for use as a sample. This can load WAVE, AIFF, RIFF, OGG, and VOC formats. Using `SDL_RWops` is not covered here, but they enable you to load from almost any source. **Note:** You must call SDL_OpenAudio before this. It must know the output characteristics so it can convert the sample for playback, it does this conversion at load time.

Returns: a pointer to the sample as a `Mix_Chunk`. **NULL** is returned on errors.

```
// load sample.wav in to sample
Mix_Chunk *sample;
sample=Mix_LoadWAV_RW(SDL_RWFromFile("sample.wav", "rb"), 1);
if(!sample) {
    printf("Mix_LoadWAV_RW: %s\n", Mix_GetError());
    // handle error
}
```

See Also:
⟨undefined⟩ [Mix_LoadWAV], page ⟨undefined⟩, ⟨undefined⟩ [Mix_QuickLoad_WAV], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FreeChunk], page ⟨undefined⟩

4.2.3 Mix_QuickLoad_WAV

`Mix_Chunk` ***Mix_QuickLoad_WAV**(`Uint8` ****mem**`)`

mem Memory buffer containing a WAVE file in output format.

Load *mem* as a WAVE/RIFF file into a new sample. The WAVE in *mem* must be already in the output format. It would be better to use `Mix_LoadWAV_RW` if you aren't sure.
Note: This function does very little checking. If the format mismatches the output format, or if the buffer is not a WAVE, it will not return an error. This is probably a dangerous function to use.

Returns: a pointer to the sample as a `Mix_Chunk`. **NULL** is returned on errors.

```
// quick-load a wave from memory
// Uint8 *wave; // I assume you have the wave loaded raw,
                // or compiled in the program...
Mix_Chunk *wave_chunk;
if(!(wave_chunk=Mix_QuickLoad_WAV(wave))) {
    printf("Mix_QuickLoad_WAV: %s\n", Mix_GetError());
    // handle error
}
```

See Also:
⟨undefined⟩ [Mix_LoadWAV], page ⟨undefined⟩, ⟨undefined⟩ [Mix_QuickLoad_RAW], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FreeChunk], page ⟨undefined⟩

4.2.4 Mix_QuickLoad_RAW

`Mix_Chunk *`**Mix_QuickLoad_RAW**`(Uint8 *`*mem*`)`

mem Memory buffer containing a WAVE file in output format.

Load *mem* as a raw sample. The data in *mem* must be already in the output format. If you aren't sure what you are doing, this is not a good function for you!
Note: This function does very little checking. If the format mismatches the output format it will not return an error. This is probably a dangerous function to use.

Returns: a pointer to the sample as a `Mix_Chunk`. **NULL** is returned on errors, such as when out of memory.

```
// quick-load a raw sample from memory
// Uint8 *raw; // I assume you have the raw data here,
                // or compiled in the program...
Mix_Chunk *raw_chunk;
if(!(raw_chunk=Mix_QuickLoad_RAW(raw))) {
    printf("Mix_QuickLoad_RAW: %s\n", Mix_GetError());
    // handle error
}
```

See Also:
⟨undefined⟩ [Mix_LoadWAV], page ⟨undefined⟩, ⟨undefined⟩ [Mix_QuickLoad_WAV], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FreeChunk], page ⟨undefined⟩

4.2.5 Mix_VolumeChunk

`int` **Mix_VolumeChunk**`(Mix_Chunk *`*`chunk`*`, int` *`volume`*`)`

chunk Pointer to the Mix_Chunk to set the volume in.

volume The volume to use from 0 to **MIX_MAX_VOLUME**(128).
If greater than **MIX_MAX_VOLUME**,
then it will be set to **MIX_MAX_VOLUME**.
If less than 0 then *chunk*->volume will not be set.

Set *chunk*->volume to *volume*.
The volume setting will take effect when the chunk is used on a channel, being mixed into the output.

Returns: previous *chunk*->volume setting. if you passed a negative value for *volume* then this volume is still the current volume for the *chunk*.

```
// set the sample's volume to 1/2
// Mix_Chunk *sample;
int previous_volume;
previous_volume=Mix_VolumeChunk(sample, MIX_MAX_VOLUME/2);
printf("previous_volume: %d\n", previous_volume);
```

See Also:
⟨undefined⟩ [Mix_Chunk], page ⟨undefined⟩

4.2.6 Mix_FreeChunk

`void` **Mix_FreeChunk**`(Mix_Chunk *`*`chunk`*`)`

chunk Pointer to the Mix_Chunk to free.

Free the memory used in *chunk*, and free *chunk* itself as well. Do not use *chunk* after this without loading a new sample to it. **Note**: It's a bad idea to free a chunk that is still being played...

```
// free the sample
// Mix_Chunk *sample;
Mix_FreeChunk(sample);
sample=NULL; // to be safe...
```

See Also:
⟨undefined⟩ [Mix_LoadWAV], page ⟨undefined⟩, ⟨undefined⟩ [Mix_LoadWAV_RW], page ⟨undefined⟩, ⟨undefined⟩ [Mix_QuickLoad_WAV], page ⟨undefined⟩,

4.3 Channels

These functions work with sound effect mixer channels. Music is not affected by these functions.

4.3.1 Mix_AllocateChannels

`int` **Mix_AllocateChannels**`(int` *`numchans`*`)`

numchans Number of channels to allocate for mixing.
A negative number will not do anything, it will tell you how many channels are currently allocated.

Set the number of channels being mixed. This can be called multiple times, even with sounds playing. If *numchans* is less than the current number of channels, then the higher channels will be stopped, freed, and therefore not mixed any longer. It's probably not a good idea to change the size 1000 times a second though.
If any channels are deallocated, any callback set by `Mix_ChannelFinished` will be called when each channel is halted to be freed. **Note**: passing in zero WILL free all mixing channels, however music will still play.

Returns: The number of channels allocated. Never fails...but a high number of channels can segfault if you run out of memory. We're talking REALLY high!

```
// allocate 16 mixing channels
Mix_AllocateChannels(16);
```

See Also:
⟨undefined⟩ [Mix_OpenAudio], page ⟨undefined⟩

4.3.2 Mix_Volume

`int` **Mix_Volume**`(int` *`channel`*`, int` *`volume`*`)`

channel Channel to set mix volume for.
`-1` will set the volume for all allocated channels.

volume The volume to use from 0 to **MIX_MAX_VOLUME**(128).
If greater than **MIX_MAX_VOLUME**,
then it will be set to **MIX_MAX_VOLUME**.
If less than 0 then the volume will not be set.

Set the volume for any allocated channel. If *channel* is `-1` then all channels at are set at once. The *volume* is applied during the final mix, along with the sample volume. So setting this volume to 64 will halve the output of all samples played on the specified channel. All channels default to a volume of 128, which is the max. Newly allocated channels will have the max volume set, so setting all channels volumes does not affect subsequent channel allocations.

Returns: current volume of the channel. If channel is -1, the average volume is returned.

```
// set channel 1 to half volume
Mix_Volume(1,MIX_MAX_VOLUME/2);

// print the average volume
printf("Average volume is %d\n",Mix_Volume(-1,-1));
```

See Also:
⟨undefined⟩ [Mix_VolumeChunk], page ⟨undefined⟩, ⟨undefined⟩ [Mix_VolumeMusic], page ⟨undefined⟩

4.3.3 Mix_PlayChannel

`int` **Mix_PlayChannel**`(int` *channel*`, Mix_Chunk *`*chunk*`, int` *loops*`)`

channel Channel to play on, or -1 for the first free unreserved channel.

chunk Sample to play.

loops Number of loops, -1 is infinite loops.
Passing one here plays the sample twice (1 loop).

Play *chunk* on *channel*, or if *channel* is -1, pick the first free unreserved channel. The sample will play for *loops*+1 number of times, unless stopped by halt, or fade out, or setting a new expiration time of less time than it would have originally taken to play the loops, or closing the mixer.
Note: this just calls `Mix_PlayChannelTimed()` with *ticks* set to -1.

Returns: the channel the sample is played on. On any errors, -1 is returned.

```
// play sample on first free unreserved channel
// play it exactly once through
// Mix_Chunk *sample; //previously loaded
if(Mix_PlayChannel(-1, sample, 0)==-1) {
    printf("Mix_PlayChannel: %s\n",Mix_GetError());
    // may be critical error, or maybe just no channels were free.
    // you could allocated another channel in that case...
}
```

See Also:
⟨undefined⟩ [Mix_PlayChannelTimed], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_HaltChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ExpireChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ReserveChannels], page ⟨undefined⟩

4.3.4 Mix_PlayChannelTimed

`int` **Mix_PlayChannelTimed**`(int` *channel*`, Mix_Chunk *`*chunk*`, int` *loops*`, int` *ticks*`)`

channel Channel to play on, or -1 for the first free unreserved channel.

chunk Sample to play.

loops Number of loops, -1 is infinite loops.
Passing one here plays the sample twice (1 loop).

ticks Millisecond limit to play sample, at most.
If not enough *loops* or the sample *chunk* is not long enough, then the sample may stop before this timeout occurs.
-1 means play forever.

If the sample is long enough and has enough loops then the sample will stop after *ticks* milliseconds. Otherwise this function is the same as ⟨undefined⟩ [Mix_PlayChannel], page ⟨undefined⟩.

Returns: the channel the sample is played on. On any errors, -1 is returned.

```
// play sample on first free unreserved channel
// play it for half a second
// Mix_Chunk *sample; //previously loaded
if(Mix_PlayChannelTimed(-1, sample, -1 , 500)==-1) {
    printf("Mix_PlayChannel: %s\n",Mix_GetError());
    // may be critical error, or maybe just no channels were free.
    // you could allocated another channel in that case...
}
```

See Also:
⟨undefined⟩ [Mix_PlayChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInChannelTimed], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ReserveChannels], page ⟨undefined⟩

4.3.5 Mix_FadeInChannel

`int` **Mix_FadeInChannel**`(int` *`channel`*`, Mix_Chunk *`*`chunk`*`, int` *`loops`*`, int` *`ms`*`)`

channel Channel to play on, or -1 for the first free unreserved channel.

chunk Sample to play.

loops Number of loops, -1 is infinite loops.
Passing one here plays the sample twice (1 loop).

ms Milliseconds of time that the fade-in effect should take to go from silence to full volume.

Play *chunk* on *channel*, or if *channel* is -1, pick the first free unreserved channel.
The channel volume starts at 0 and fades up to full volume over *ms* milliseconds of time. The sample may end before the fade-in is complete if it is too short or doesn't have enough loops. The sample will play for *loops*+1 number of times, unless stopped by halt, or fade out, or setting a new expiration time of less time than it would have originally taken to play the loops, or closing the mixer.
Note: this just calls `Mix_FadeInChannelTimed()` with *ticks* set to -1.

Returns: the channel the sample is played on. On any errors, -1 is returned.

```
// play sample on first free unreserved channel
// play it exactly 3 times through
// fade in over one second
// Mix_Chunk *sample; //previously loaded
if(Mix_FadeInChannel(-1, sample, 2, 1000)==-1) {
    printf("Mix_FadeInChannel: %s\n",Mix_GetError());
    // may be critical error, or maybe just no channels were free.
    // you could allocated another channel in that case...
}
```

See Also:
⟨undefined⟩ [Mix_PlayChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInChannelTimed], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadingChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ReserveChannels], page ⟨undefined⟩

4.3.6 Mix_FadeInChannelTimed

`int` **Mix_FadeInChannelTimed**`(int` *channel*`, Mix_Chunk *`*chunk*`, int` *loops*`, int` *ms*`, int` *ticks*`)`

channel Channel to play on, or -1 for the first free unreserved channel.

chunk Sample to play.

loops Number of loops, -1 is infinite loops.
Passing one here plays the sample twice (1 loop).

ms Milliseconds of time that the fade-in effect should take to go from silence to full volume.

ticks Millisecond limit to play sample, at most.
If not enough *loops* or the sample *chunk* is not long enough, then the sample may stop before this timeout occurs.
-1 means play forever.

If the sample is long enough and has enough loops then the sample will stop after *ticks* milliseconds. Otherwise this function is the same as ⟨undefined⟩ [Mix_FadeInChannel], page ⟨undefined⟩.

Returns: the channel the sample is played on. On any errors, -1 is returned.

```
// play sample on first free unreserved channel
// play it for half a second
// Mix_Chunk *sample; //previously loaded
if(Mix_PlayChannelTimed(-1, sample, -1 , 500)==-1) {
    printf("Mix_PlayChannel: %s\n",Mix_GetError());
    // may be critical error, or maybe just no channels were free.
    // you could allocated another channel in that case...
}
```

See Also:
⟨undefined⟩ [Mix_PlayChannelTimed], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadingChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_HaltChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ExpireChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ReserveChannels], page ⟨undefined⟩

4.3.7 Mix_Pause

`void` **Mix_Pause**`(int` *`channel`*`)`

channel Channel to pause on, or -1 for all channels.

Pause *channel*, or all playing channels if -1 is passed in. You may still halt a paused channel.
Note: Only channels which are actively playing will be paused.

```
// pause all sample playback
Mix_Pause(-1);
```

See Also:
⟨undefined⟩ [Mix_Resume], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Paused], page ⟨undefined⟩, ⟨undefined⟩ [Mix_HaltChannel], page ⟨undefined⟩

4.3.8 Mix_Resume

`void` **Mix_Resume**`(int` *`channel`*`)`

channel Channel to resume playing, or -1 for all channels.

Unpause *channel*, or all playing and paused channels if -1 is passed in.

```
// resume playback on all previously active channels
Mix_Resume(-1);
```

See Also:
⟨undefined⟩ [Mix_Pause], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Paused], page ⟨undefined⟩

4.3.9 Mix_HaltChannel

`int` **Mix_HaltChannel**`(int` *`channel`*`)`

channel Channel to stop playing, or -1 for all channels.

Halt *channel* playback, or all channels if -1 is passed in.
Any callback set by `Mix_ChannelFinished` will be called.

Returns: always returns zero. (kinda silly)

```
// halt playback on all channels
Mix_HaltChannel(-1);
```

See Also:
⟨undefined⟩ [Mix_ExpireChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ChannelFinished], page ⟨undefined⟩

4.3.10 Mix_ExpireChannel

`int` **Mix_ExpireChannel**(`int` *`channel`*, `int` *`ticks`*)

channel Channel to stop playing, or -1 for all channels.

ticks Millisecons until channel(s) halt playback.

Halt *channel* playback, or all channels if -1 is passed in, after *ticks* milliseconds. Any callback set by `Mix_ChannelFinished` will be called when the channel expires.

Returns: Number of channels set to expire. Whether or not they are active.

```
// halt playback on all channels in 2 seconds
Mix_ExpireChannel(-1, 2000);
```

See Also:
⟨undefined⟩ [Mix_HaltChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ChannelFinished], page ⟨undefined⟩

4.3.11 Mix_FadeOutChannel

`int` **Mix_FadeOutChannel**(`int` *`channel`*, `int` *`ms`*)

channel Channel to fade out, or -1 to fade all channels out.

ms Milliseconds of time that the fade-out effect should take to go to silence, starting now.

Gradually fade out *which* channel over *ms* milliseconds starting from now. The channel will be halted after the fade out is completed. Only channels that are playing are set to fade out, including paused channels. Any callback set by `Mix_ChannelFinished` will be called when the channel finishes fading out.

Returns: The number of channels set to fade out.

```
// fade out all channels to finish 3 seconds from now
printf("starting fade out of %d channels\n", Mix_FadeOutChannel(-1, 3000));
```

See Also:
⟨undefined⟩ [Mix_FadeInChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInChannelTimed], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadingChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ChannelFinished], page ⟨undefined⟩

4.3.12 Mix_ChannelFinished

`void` **Mix_ChannelFinished**`(void (*`*`channel_finished`*`)(int channel))`

channel_finished
: Function to call when any channel finishes playback.

When *channel* playback is halted, then the specified *channel_finished* function is called. The `channel` parameter will contain the channel number that has finished.
NOTE: NEVER call SDL_Mixer functions, nor `SDL_LockAudio`, from a callback function.

```
// a simple channel_finished function
void channelDone(int channel) {
    printf("channel %d finished playback.\n",channel);
}
```

```
// make a channelDone function
void channelDone(int channel)
{
    printf("channel %d finished playing.\n", channel);
}
...
// set the callback for when a channel stops playing
Mix_ChannelFinished(channelDone);
```

See Also:
⟨undefined⟩ [Mix_HaltChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ExpireChannel], page ⟨undefined⟩

4.3.13 Mix_Playing

`int` **Mix_Playing**`(int` *`channel`*`)`

channel Channel to test whether it is playing or not.
-1 will tell you how many channels are playing.

Tells you if *channel* is playing, or not.
Note: Does not check if the channel has been paused.

Returns: Zero if the channel is not playing. Otherwise if you passed in -1, the number of channels playing is returned. If you passed in a specific channel, then 1 is returned if it is playing.

```
// check how many channels are playing samples
printf("%d channels are playing\n", Mix_Playing(-1));
```

See Also:
⟨undefined⟩ [Mix_Paused], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Fading], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PlayChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Pause], page ⟨undefined⟩,

4.3.14 Mix_Paused

`int` **Mix_Paused**`(int` *`channel`*`)`

channel Channel to test whether it is paused or not.
-1 will tell you how many channels are paused.

Tells you if *channel* is paused, or not.
Note: Does not check if the channel has been halted after it was paused, which may seem a little weird.

Returns: Zero if the channel is not paused. Otherwise if you passed in -1, the number of paused channels is returned. If you passed in a specific channel, then 1 is returned if it is paused.

```
// check the pause status on all channels
printf("%d channels are paused\n", Mix_Paused(-1));
```

See Also:
⟨undefined⟩ [Mix_Playing], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Pause], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Resume], page ⟨undefined⟩

4.3.15 Mix_FadingChannel

`Mix_Fading` **Mix_FadingChannel**(`int` *which*)

which Channel to get the fade activity status from.
-1 is not valid, and will probably crash the program.

Tells you if *which* channel is fading in, out, or not. Does not tell you if the channel is playing anything, or paused, so you'd need to test that separately.

Returns: the fading status. Never returns an error.

```
// check the fade status on channel 0
switch(Mix_FadingChannel(0)) {
    case MIX_NO_FADING:
        printf("Not fading.\n");
        break;
    case MIX_FADING_OUT:
        printf("Fading out.\n");
        break;
    case MIX_FADING_IN:
        printf("Fading in.\n");
        break;
}
```

See Also:
⟨undefined⟩ [Mix_Fading], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Playing], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Paused], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInChannelTimed], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutChannel], page ⟨undefined⟩

4.3.16 Mix_GetChunk

`Mix_Chunk *`**Mix_GetChunk**`(int` *`channel`*`)`

channel Channel to get the current Mix_Chunk playing.
-1 is not valid, but will not crash the program.

Get the most recent sample chunk pointer played on *channel*. This pointer may be currently playing, or just the last used.
Note: The actual chunk may have been freed, so this pointer may not be valid anymore.

Returns: Pointer to the Mix_Chunk. NULL is returned if the channel is not allocated, or if the channel has not played any samples yet.

```
// get the last chunk used by channel 0
printf("Mix_Chunk* last in use on channel 0 was: %08p\n", Mix_GetChunk(0));
```

See Also:
⟨undefined⟩ [Mix_Chunk], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Playing], page ⟨undefined⟩

4.4 Groups

These functions work with groupings of mixer channels.

The default group tag number of -1, which refers to ALL channels.

4.4.1 Mix_ReserveChannels

`int` **Mix_ReserveChannels**`(int` *num*`)`

num Number of channels to reserve from default mixing.
Zero removes all reservations.

Reserve *num* channels from being used when playing samples when passing in -1 as a channel number to playback functions. The channels are reserved starting from channel 0 to *num*-1. Passing in zero will unreserve all channels. Normally SDL_mixer starts without any channels reserved.

The following functions are affected by this setting:
⟨undefined⟩ [Mix_PlayChannel], page ⟨undefined⟩
⟨undefined⟩ [Mix_PlayChannelTimed], page ⟨undefined⟩
⟨undefined⟩ [Mix_FadeInChannel], page ⟨undefined⟩
⟨undefined⟩ [Mix_FadeInChannelTimed], page ⟨undefined⟩

Returns: The number of channels reserved. Never fails, but may return less channels than you ask for, depending on the number of channels previously allocated.

```
// reserve the first 8 mixing channels
int reserved_count;
reserved_count=Mix_ReserveChannels(8);
if(reserved_count!=8) {
    printf("reserved %d channels from default mixing.\n",reserved_count);
    printf("8 channels were not reserved!\n");
    // this might be a critical error...
}
```

See Also:
⟨undefined⟩ [Mix_AllocateChannels], page ⟨undefined⟩

4.4.2 Mix_GroupChannel

`int` **Mix_GroupChannel**`(int` *`which`*`, int` *`tag`*`)`

which Channel number of channels to assign *tag* to.

tag A group number Any positive numbers (including zero).
-1 is the default group. Use -1 to remove a group tag essentially.

Add *which* channel to group *tag*, or reset it's group to the default group tag (-1).

Returns: True(1) on success. False(0) is returned when the channel specified is invalid.

```
// add channel 0 to group 1
if(!Mix_GroupChannel(0,1)) {
    // bad channel, apparently channel 1 isn't allocated
}
```

See Also:
⟨undefined⟩ [Mix_GroupChannels], page ⟨undefined⟩, ⟨undefined⟩ [Mix_AllocateChannels], page ⟨undefined⟩

4.4.3 Mix_GroupChannels

`int` **Mix_GroupChannels**`(int` *`from`*`, int` *`to`*`, int` *`tag`*`)`

from	First Channel number of channels to assign *tag* to. Must be less or equal to *to*.
to	Last Channel number of channels to assign *tag* to. Must be greater or equal to *from*.
tag	A group number. Any positive numbers (including zero). -1 is the default group. Use -1 to remove a group tag essentially.

Add channels starting at *from* up through *to* to group *tag*, or reset it's group to the default group tag (-1).

Returns: The number of tagged channels on success. If that number is less than *to*-*from*+1 then some channels were no tagged because they didn't exist.

```
// add channels 0 through 7 to group 1
if(Mix_GroupChannels(0,7,1)!=8) {
    // some bad channels, apparently some channels aren't allocated
}
```

See Also:
⟨undefined⟩ [Mix_GroupChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_AllocateChannels], page ⟨undefined⟩

4.4.4 Mix_GroupCount

`int` **Mix_GroupCount**`(int` *tag*`)`

tag A group number Any positive numbers (including zero).
-1 will count ALL channels.

Count the number of channels in group *tag*.

Returns: The number of channels found in the group. This function never fails.

```
// count the number of channels in group 1
printf("There are %d channels in group 1\n", Mix_GroupCount(1));
```

See Also:
⟨undefined⟩ [Mix_GroupChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupChannels], page ⟨undefined⟩

4.4.5 Mix_GroupAvailable

`int` **Mix_GroupAvailable**`(int` *tag*`)`

tag A group number Any positive numbers (including zero).
-1 will search ALL channels.

Find the first available (not playing) channel in group *tag*.

Returns: The channel found on success. -1 is returned when no channels in the group are available.

```
// find the first available channel in group 1
int channel;
channel=Mix_GroupAvailable(1);
if (channel==-1) {
    // no channel available...
    // perhaps search for oldest or newest channel in use...
}
```

See Also:
⟨undefined⟩ [Mix_GroupOldest], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupNewer], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupChannels], page ⟨undefined⟩

4.4.6 Mix_GroupOldest

`int` **Mix_GroupOldest**`(int` *tag*`)`

tag A group number Any positive numbers (including zero).
-1 will search ALL channels.

Find the oldest actively playing channel in group *tag*.

Returns: The channel found on success. -1 is returned when no channels in the group are playing or the group is empty.

```
// find the oldest playing channel in group 1
int channel;
channel=Mix_GroupOldest(1);
if (channel==-1) {
    // no channel playing or allocated...
    // perhaps just search for an available channel...
}
```

See Also:
⟨undefined⟩ [Mix_GroupNewer], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupAvailable], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupChannels], page ⟨undefined⟩

4.4.7 Mix_GroupNewer

`int` **Mix_GroupNewer**`(int` *tag*`)`

tag A group number Any positive numbers (including zero).
-1 will search ALL channels.

Find the newest, most recently started, actively playing channel in group *tag*.

Returns: The channel found on success. -1 is returned when no channels in the group are playing or the group is empty.

```
// find the newest playing channel in group 1
int channel;
channel=Mix_GroupNewer(1);
if (channel==-1) {
    // no channel playing or allocated...
    // perhaps just search for an available channel...
}
```

See Also:
⟨undefined⟩ [Mix_GroupOldest], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupAvailable], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GroupChannels], page ⟨undefined⟩

4.4.8 Mix_FadeOutGroup

`int` **Mix_FadeOutGroup**`(int` *tag*`, int` *ms*`)`

tag Group to fade out.
NOTE: -1 will **NOT** fade all channels out. Use `Mix_FadeOutChannel(-1)` for that instead.

ms Milliseconds of time that the fade-out effect should take to go to silence, starting now.

Gradually fade out channels in group *tag* over *ms* milliseconds starting from now. The channels will be halted after the fade out is completed. Only channels that are playing are set to fade out, including paused channels. Any callback set by `Mix_ChannelFinished` will be called when each channel finishes fading out.

Returns: The number of channels set to fade out.

```
// fade out all channels in group 1 to finish 3 seconds from now
printf("starting fade out of %d channels\n", Mix_FadeOutGroup(1, 3000));
```

See Also:
⟨undefined⟩ [Mix_HaltGroup], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadingChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ChannelFinished], page ⟨undefined⟩

4.4.9 Mix_HaltGroup

`int` **Mix_HaltGroup**`(int` *tag*`)`

tag Group to fade out.
NOTE: -1 will **NOT** halt all channels. Use `Mix_HaltChannel(-1)` for that instead.

Halt playback on all channels in group *tag*.
Any callback set by `Mix_ChannelFinished` will be called once for each channel that stops.

Returns: always returns zero. (more silly than *Mix_HaltChannel*)

```
// halt playback on all channels in group 1
Mix_HaltGroup(1);
```

See Also:
⟨undefined⟩ [Mix_FadeOutGroup], page ⟨undefined⟩, ⟨undefined⟩ [Mix_HaltChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ChannelFinished], page ⟨undefined⟩

4.5 Music

These functions work with music. Music is not played on a normal mixer channel. Music is therefore manipulated separately, except in post-processing hooks.

4.5.1 Mix_LoadMUS

`Mix_Music *`**Mix_LoadMUS**`(const char *`*`file`*`)`

file Name of music file to use.

Load music file to use. This can load WAVE, MOD, MIDI, OGG, MP3, and any file that you use a command to play with.
If you are using an external command to play the music, you must call `Mix_SetMusicCMD` before this, otherwise the internal players will be used. Alternatively, if you have set an external command up and don't want to use it, you must call `Mix_SetMusicCMD(NULL)` to use the built-in players again.

Returns: A pointer to a Mix_Music. NULL is returned on errors.

```
// load the MP3 file "music.mp3" to play as music
Mix_Music *music;
music=Mix_LoadMUS("music.mp3");
if(!music) {
    printf("Mix_LoadMUS(\"music.mp3\"): %s\n", Mix_GetError());
    // this might be a critical error...
}
```

See Also:
⟨undefined⟩ [Mix_Music], page ⟨undefined⟩, ⟨undefined⟩ [Mix_SetMusicCMD], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PlayMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInMusicPos], page ⟨undefined⟩

4.5.2 Mix_FreeMusic

`void` **Mix_FreeMusic**`(Mix_Music *`*music*`)`

music Pointer to Mix_Music to free.

Free the loaded *music*. If *music* is playing it will be halted. If *music* is fading out, then this function will wait (blocking) until the fade out is complete.

```
// free music
Mix_Music *music;
Mix_FreeMusic(music);
music=NULL; // so we know we freed it...
```

See Also:
⟨undefined⟩ [Mix_LoadMUS], page ⟨undefined⟩

4.5.3 Mix_PlayMusic

`int` **Mix_PlayMusic**`(Mix_Music *`*`music`*`, int `*`loops`*`)`

music Pointer to Mix_Music to play.

loops number of times to play through the music.
0 plays the music zero times...
-1 plays the music forever (or as close as it can get to that)

Play the loaded *music* *loop* times through from start to finish. The previous music will be halted, or if fading out it waits (blocking) for that to finish.

Returns: 0 on success, or -1 on errors.

```
// play music forever
// Mix_Music *music; // I assume this has been loaded already
if(Mix_PlayMusic(music, -1)==-1) {
    printf("Mix_PlayMusic: %s\n", Mix_GetError());
    // well, there's no music, but most games don't break without music...
}
```

See Also:
⟨undefined⟩ [Mix_FadeInMusic], page ⟨undefined⟩

4.5.4 Mix_FadeInMusic

`int` **Mix_FadeInMusic**`(Mix_Music *`*music*`, int` *loops*`, int` *ms*`)`

music Pointer to Mix_Music to play.

loops number of times to play through the music.
0 plays the music zero times...
-1 plays the music forever (or as close as it can get to that)

ms Milliseconds for the fade-in effect to complete.

Fade in over *ms* milliseconds of time, the loaded *music*, playing it *loop* times through from start to finish.
The fade in effect only applies to the first loop.
Any previous music will be halted, or if it is fading out it will wait (blocking) for the fade to complete.
This function is the same as `Mix_FadeInMusicPos(`*music*`,` *loops*`,` *ms*`, 0)`.

Returns: 0 on success, or -1 on errors.

```
// play music forever, fading in over 2 seconds
// Mix_Music *music; // I assume this has been loaded already
if(Mix_FadeInMusic(music, -1, 2000)==-1) {
    printf("Mix_FadeInMusic: %s\n", Mix_GetError());
    // well, there's no music, but most games don't break without music...
}
```

See Also:
⟨undefined⟩ [Mix_PlayMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInMusicPos], page ⟨undefined⟩

4.5.5 Mix_FadeInMusicPos

`int` **Mix_FadeInMusicPos**`(Mix_Music *`*music*`, int` *loops*`, int` *ms*`, double` *position*`)`

music Pointer to Mix_Music to play.

loops number of times to play through the music.
0 plays the music zero times...
-1 plays the music forever (or as close as it can get to that)

ms Milliseconds for the fade-in effect to complete.

position Posistion to play from, see `Mix_SetMusicPosition` for meaning.

Fade in over *ms* milliseconds of time, the loaded *music*, playing it *loop* times through from start to finish.
The fade in effect only applies to the first loop.
The first time the music is played, it posistion will be set to *posistion*, which means different things for different types of music files, see `Mix_SetMusicPosition` for more info on that.
Any previous music will be halted, or if it is fading out it will wait (blocking) for the fade to complete.

Returns: 0 on success, or -1 on errors.

```
// play music forever, fading in over 2 seconds
// Mix_Music *music; // I assume this has been loaded already
if(Mix_FadeInMusicPos(music, -1, 2000)==-1) {
    printf("Mix_FadeInMusic: %s\n", Mix_GetError());
    // well, there's no music, but most games don't break without music...
}
```

See Also:
⟨undefined⟩ [Mix_PlayMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_SetMusicPosition], page ⟨undefined⟩

4.5.6 Mix_HookMusic

`void` **Mix_HookMusic**`(void (*`*mix_func*`)(void *udata, Uint8 *stream, int len), void *`*arg*`)`

mix_func Function pointer to a music player mixer function.
NULL will stop the use of the music player, returning the mixer to using the internal music players like usual.

arg This is passed to the *mix_func*'s `udata` parameter when it is called.

This sets up a custom music player function. The function will be called with *arg* passed into the `udata` parameter when the *mix_func* is called. The `stream` parameter passes in the audio stream buffer to be filled with `len` bytes of music. The music player will then be called automatically when the mixer needs it. Music playing will start as soon as this is called. All the music playing and stopping functions have no effect on music after this. Pause and resume will work. Using a custom music player and the internal music player is not possible, the custom music player takes priority. To stop the custom music player call `Mix_HookMusic(NULL, NULL)`.
NOTE: NEVER call SDL_Mixer functions, nor `SDL_LockAudio`, from a callback function.

```
// make a music play function
// it expects udata to be a pointer to an int
void myMusicPlayer(void *udata, Uint8 *stream, int len)
{
    int i, pos=*(int*)udata;

    // fill buffer with...uh...music...
    for(i=0; i<len; i++)
        stream[i]=(i+pos)&ff;

    // set udata for next time
    pos+=len;
    *(int*)udata=pos;
}
...
// use myMusicPlayer for playing...uh...music
int music_pos=0;
Mix_HookMusic(myMusicPlayer, &music_pos);
```

See Also:
⟨undefined⟩ [Mix_SetMusicCMD], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GetMusicHookData], page ⟨undefined⟩

4.5.7 Mix_VolumeMusic

`int` **Mix_VolumeMusic**`(int` *volume*`)`

volume Music volume, from 0 to **MIX_MAX_VOLUME**(128).
Values greater than **MIX_MAX_VOLUME** will use **MIX_MAX_VOLUME**.
-1 does not set the volume, but does return the current volume setting.

Set the volume to *volume*, if it is 0 or greater, and return the previous volume setting. Setting the volume during a fade will not work, the faders use this function to perform their effect! Setting volume while using an external music player set by `Mix_SetMusicCMD` will have no effect, and `Mix_GetError` will show the reason why not.

Returns: The previous volume setting.

```
// set the music volume to 1/2 maximum, and then check it
printf("volume was    : %d\n", Mix_VolumeMusic(MIX_MAX_VOLUME/2));
printf("volume is now : %d\n", Mix_VolumeMusic(-1));
```

See Also:
⟨undefined⟩ [Mix_FadeInMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_SetMusicCMD], page ⟨undefined⟩

4.5.8 Mix_PauseMusic

`void` **Mix_PauseMusic**`()`

Pause the music playback. You may halt paused music.
Note: Music can only be paused if it is actively playing.

```
// pause music playback
Mix_PauseMusic();
```

See Also:
⟨undefined⟩ [Mix_ResumeMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PausedMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_HaltMusic], page ⟨undefined⟩

4.5.9 Mix_ResumeMusic

`void` **Mix_ResumeMusic**`()`

Unpause the music. This is safe to use on halted, paused, and already playing music.

```
// resume music playback
Mix_ResumeMusic();
```

See Also:
⟨undefined⟩ [Mix_PauseMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PausedMusic], page ⟨undefined⟩

4.5.10 Mix_RewindMusic

`void` **Mix_RewindMusic**`()`

Rewind the music to the start. This is safe to use on halted, paused, and already playing music. It is not useful to rewind the music immediately after starting playback, because it starts at the beginning by default.

This function only works for these streams: MOD, OGG, MP3, Native MIDI.

```
// rewind music playback to the start
Mix_RewindMusic();
```

See Also:
⟨undefined⟩ [Mix_PlayMusic], page ⟨undefined⟩

4.5.11 Mix_SetMusicPosition

`int` **Mix_SetMusicPosition**`(double` *`position`*`)`

position Posistion to play from.

Set the position of the currently playing music. The *position* takes different meanings for different music sources. It only works on the music sources listed below.

MOD The double is cast to Uint16 and used for a pattern number in the module. Passing zero is similar to rewinding the song.

OGG Jumps to *position* seconds from the beginning of the song.

MP3 Jumps to *position* seconds from the current position in the stream. So you may want to call `Mix_RewindMusic` before this. Does not go in reverse...negative values do nothing.

Returns: 0 on success, or -1 if the codec doesn't support this function.

```
// skip one minute into the song, from the start
// this assumes you are playing an MP3
Mix_RewindMusic();
if(Mix_SetMusicPosition(60.0)==-1) {
    printf("Mix_SetMusicPosition: %s\n", Mix_GetError());
}
```

See Also:
⟨undefined⟩ [Mix_FadeInMusicPos], page ⟨undefined⟩

4.5.12 Mix_SetMusicCMD

`int` **Mix_SetMusicCMD**`(const char *`*command*`)`

command System command to play the music. Should be a complete command, as if typed in to the command line, but it should expect the filename to be added as the last argument.
NULL will turn off using an external command for music, returning to the internal music playing functionality.

Setup a command line music player to use to play music. Any music playing will be halted. The music file to play is set by calling `Mix_LoadMUS(filename)`, and the filename is appended as the last argument on the commandline. This allows you to reuse the music command to play multiple files. The command will be sent signals **SIGTERM** to halt, **SIGSTOP** to pause, and **SIGCONT** to resume. The command program should react correctly to those signals for it to function properly with SDL_Mixer. `Mix_VolumeMusic` has no effect when using an external music player, and `Mix_GetError` will have an error code set. You should set the music volume in the music player's command if the music player supports that. Looping music works, by calling the command again when the previous music player process has ended. Playing music through a command uses a forked process to execute the music command.
To use the internal music players set the *command* to **NULL**.
NOTE: External music is not mixed by SDL_mixer, so no post-processing hooks will be for music.
NOTE: Playing music through an external command may not work if the sound driver does not support multiple openings of the audio device, since SDL_Mixer already has the audio device open for playing samples through channels.
NOTE: Commands are not totally portable, so be careful.

Returns: 0 on success, or -1 on any errors, such as running out of memory.

```
// use mpg123 to play music
Mix_Music *music=NULL;
if(Mix_SetMusicCMD("mpg123 -q")==-1) {
    perror("Mix_SetMusicCMD");
} else {
    // play some mp3 file
    music=Mix_LoadMUS("music.mp3");
    if(music) {
        Mix_PlayMusic(music,1);
    }
}
```

See Also:
⟨undefined⟩ [Mix_PlayMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_VolumeMusic], page ⟨undefined⟩

4.5.13 Mix_HaltMusic

`int` **Mix_HaltMusic**`()`

Halt playback of music. This interrupts music fader effects. Any callback set by `Mix_HookMusicFinished` will be called when the music stops.

Returns: always returns zero. (even more silly than *Mix_HaltGroup*)

```
// halt music playback
Mix_HaltMusic();
```

See Also:
⟨undefined⟩ [Mix_FadeOutMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_HookMusicFinished], page ⟨undefined⟩

4.5.14 Mix_FadeOutMusic

`int` **Mix_FadeOutMusic**`(int` *ms*`)`

ms Milliseconds of time that the fade-out effect should take to go to silence, starting now.

Gradually fade out the music over *ms* milliseconds starting from now. The music will be halted after the fade out is completed. Only when music is playing and not fading already are set to fade out, including paused channels. Any callback set by `Mix_HookMusicFinished` will be called when the music finishes fading out.

Returns: 1 on success, 0 on failure.

```
// fade out music to finish 3 seconds from now
while(!Mix_FadeOutMusic(3000) && Mix_PlayingMusic()) {
    // wait for any fades to complete
    SDL_Delay(100);
}
```

See Also:
⟨undefined⟩ [Mix_HaltMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadingMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PlayingMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_HookMusicFinished], page ⟨undefined⟩

4.5.15 Mix_HookMusicFinished

`void` **Mix_HookMusicFinished**`(void (*`*`music_finished`*`)())`

music_finished
: Function pointer to a `void function()`.
NULL will remove the hook.

This sets up a function to be called when music playback is halted. Any time music stops, the *music_finished* function will be called. Call with **NULL** to remove the callback.
NOTE: NEVER call SDL_Mixer functions, nor `SDL_LockAudio`, from a callback function.

```
// make a music finished function
void musicFinished()
{
    printf("Music stopped.\n");
}
...
// use musicFinished for when music stops
Mix_HookMusicFinished(musicFinished);
```

See Also:
⟨undefined⟩ [Mix_HaltMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutMusic], page ⟨undefined⟩

4.5.16 Mix_GetMusicType

`Mix_MusicType` **Mix_GetMusicType**(`const Mix_Music *`*music*`)`

music The music to get the type of.
NULL will get the currently playing music type.

Tells you the file format encoding of the music. This may be handy when used with `Mix_SetMusicPosition`, and other music functions that vary based on the type of music being played. If you want to know the type of music currently being played, pass in **NULL** to *music*.

Returns: The type of *music* or if *music* is **NULL** then the currently playing music type, otherwise **MUS_NONE** if no music is playing.

```
// print the type of music currently playing
switch(Mix_GetMusicType(NULL))
{
    case MUS_NONE:
    MUS_CMD:
        printf("Command based music is playing.\n");
        break;
    MUS_WAV:
        printf("WAVE/RIFF music is playing.\n");
        break;
    MUS_MOD:
        printf("MOD music is playing.\n");
        break;
    MUS_MID:
        printf("MIDI music is playing.\n");
        break;
    MUS_OGG:
        printf("OGG music is playing.\n");
        break;
    MUS_MP3:
        printf("MP3 music is playing.\n");
        break;
    default:
        printf("Unknown music is playing.\n");
        break;
}
```

See Also:
⟨undefined⟩ [Mix_MusicType], page ⟨undefined⟩, ⟨undefined⟩ [Mix_SetPosition], page ⟨undefined⟩

4.5.17 Mix_PlayingMusic

`int` **Mix_PlayingMusic**`()`

Tells you if music is actively playing, or not.
Note: Does not check if the channel has been paused.

Returns: Zero if the music is not playing, or 1 if it is playing.

```
// check if music is playing
printf("music is%s playing.\n", Mix_PlayingMusic()?"":" not");
```

See Also:
⟨undefined⟩ [Mix_PausedMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadingMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PlayMusic], page ⟨undefined⟩

4.5.18 Mix_PausedMusic

`int` **Mix_PausedMusic**`()`

Tells you if music is paused, or not.
Note: Does not check if the music was been halted after it was paused, which may seem a little weird.

Returns: Zero if music is not paused. 1 if it is paused.

```
// check the music pause status
printf("music is%s paused\n", Mix_PausedMusic()?"":" not");
```

See Also:
⟨undefined⟩ [Mix_PlayingMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PauseMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_ResumeMusic], page ⟨undefined⟩

4.5.19 Mix_FadingMusic

`Mix_Fading` **Mix_FadingMusic**`()`

Tells you if music is fading in, out, or not at all. Does not tell you if the channel is playing anything, or paused, so you'd need to test that separately.

Returns: the fading status. Never returns an error.

```
// check the music fade status
switch(Mix_FadingMusic()) {
    case MIX_NO_FADING:
        printf("Not fading music.\n");
        break;
    case MIX_FADING_OUT:
        printf("Fading out music.\n");
        break;
    case MIX_FADING_IN:
        printf("Fading in music.\n");
        break;
}
```

See Also:
⟨undefined⟩ [Mix_Fading], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PausedMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_PlayingMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeInMusicPos], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadeOutMusic], page ⟨undefined⟩

4.5.20 Mix_GetMusicHookData

`void` ***Mix_GetMusicHookData**`()`

Get the *arg* passed into `Mix_HookMusic`.

Returns: the *arg* pointer.

```
// retrieve the music hook data pointer
void *data;
data=Mix_GetMusicHookData();
```

See Also:
⟨undefined⟩ [Mix_HookMusic], page ⟨undefined⟩

4.6 Effects

These functions are for special effects processing. Not all effects are all that special. All effects are post processing routines that are either built-in to SDL_mixer or created by you. Effects can be applied to individual channels, or to the final mixed stream which contains all the channels including music.

The built-in processors: `Mix_SetPanning`, `Mix_SetPosition`, `Mix_SetDistance`, and `Mix_SetReverseStereo`, all look for an environment variable, **MIX_EFFECTSMAXSPEED** to be defined. If the environment variable is defined these processors may use more memory or reduce the quality of the effects, all for better speed.

4.6.1 Mix_RegisterEffect

`int` **Mix_RegisterEffect**`(int` *chan*`, Mix_EffectFunc_t` *f*`, Mix_EffectDone_t` *d*`, void *`*arg*`)`

chan channel number to register *f* and *d* on.
Use **MIX_CHANNEL_POST** to process the postmix stream.

f The function pointer for the effects processor.

d The function pointer for any cleanup routine to be called when the channel is done playing a sample.
This may be **NULL** for any processors that don't need to clean up any memory or other dynamic data.

arg A pointer to data to pass into the *f*'s and *d*'s `udata` parameter. It is a good place to keep the state data for the processor, especially if the processor is made to handle multiple channels at the same time.
This may be **NULL**, depending on the processor.

Hook a processor function *f* into a channel for post processing effects. You may just be reading the data and displaying it, or you may be altering the stream to add an echo. Most processors also have state data that they allocate as they are in use, this would be stored in the *arg* pointer data space. When a processor is finished being used, any function passed into *d* will be called, which is when your processor should clean up the data in the *arg* data space.
The effects are put into a linked list, and always appended to the end, meaning they always work on previously registered effects output. Effects may be added multiple times in a row. Effects are cumulative this way.

Returns: Zero on errors, such as a nonexisting channel.

```
// make a passthru processor function that does nothing...
void noEffect(int chan, void *stream, int len, void *udata)
{
    // you could work with stream here...
}
...
// register noEffect as a postmix processor
if(!Mix_RegisterEffect(MIX_CHANNEL_POST, noEffect, NULL, NULL)) {
    printf("Mix_RegisterEffect: %s\n", Mix_GetError());
}
```

See Also:
⟨undefined⟩ [Mix_UnregisterEffect], page ⟨undefined⟩, ⟨undefined⟩ [Mix_UnregisterAllEffects], page ⟨undefined⟩

4.6.2 Mix_UnregisterEffect

`int` **Mix_UnregisterEffect**`(int` *`channel`*`, Mix_EffectFunc_t` *`f`*`)`

channel Channel number to remove *f* from as a post processor.
Use **MIX_CHANNEL_POST** for the postmix stream.

f The function to remove from *channel*.

Remove the oldest (first found) registered effect function *f* from the effect list for *channel*. This only removes the first found occurance of that function, so it may need to be called multiple times if you added the same function multiple times, just stop removing when `Mix_UnregisterEffect` returns an error, to remove all occurances of *f* from a channel.
If the channel is active the registered effect will have its `Mix_EffectDone_t` function called, if it was specified in `Mix_RegisterEffect`.

Returns: Zero on errors, such as invalid channel, or effect function not registered on channel.

```
// unregister the noEffect from the postmix effects
// this removes all occurances of noEffect registered to the postmix
while(Mix_UnregisterEffect(MIX_CHANNEL_POST, noEffect));
// you may print Mix_GetError() if you want to check it.
// it should say "No such effect registered" after this loop.
```

See Also:
⟨undefined⟩ [Mix_UnregisterAllEffects], page ⟨undefined⟩, ⟨undefined⟩ [Mix_RegisterEffect], page ⟨undefined⟩

4.6.3 Mix_UnregisterAllEffects

`int` **Mix_UnregisterAllEffects**`(int` *`channel`*`)`

channel Channel to remove all effects from.
Use **MIX_CHANNEL_POST** for the postmix stream.

This removes all effects registered to *channel*. If the channel is active all the registered effects will have their `Mix_EffectDone_t` functions called, if they were specified in `Mix_RegisterEffect`.

Returns: Zero on errors, such as *channel* not existing.

```
// remove all effects from channel 0
if(!Mix_UnregisterAllEffects(0)) {
    printf("Mix_UnregisterAllEffects: %s\n", Mix_GetError());
}
```

See Also:
⟨undefined⟩ [Mix_UnregisterEffect], page ⟨undefined⟩, ⟨undefined⟩ [Mix_RegisterEffect], page ⟨undefined⟩

4.6.4 Mix_SetPostMix

`void` **Mix_SetPostMix**`(void (*`*mix_func*`)(void *udata, Uint8 *stream, int len), void *`*arg*`)`

mix_func The function pointer for the postmix processor.
NULL unregisters the current postmixer.

arg A pointer to data to pass into the *mix_func*'s `udata` parameter. It is a good place to keep the state data for the processor, especially if the processor is made to handle multiple channels at the same time.
This may be **NULL**, depending on the processor.

Hook a processor function *mix_func* to the postmix stream for post processing effects. You may just be reading the data and displaying it, or you may be altering the stream to add an echo. Most processors also have state data that they allocate as they are in use, this would be stored in the *arg* pointer data space. This processor is never really finished, until the audio device is closed, or you pass **NULL** as the *mix_func*.
There can only be one postmix function used at a time through this method. Use `Mix_RegisterEffect(MIX_CHANNEL_POST, mix_func, NULL, arg)` to use multiple postmix processors.
This postmix processor is run AFTER all the registered postmixers set up by `Mix_RegisterEffect`.

```
// make a passthru processor function that does nothing...
void noEffect(void *udata, Uint8 *stream, int len)
{
    // you could work with stream here...
}
...
// register noEffect as a postmix processor
Mix_SetPostMix(noEffect, NULL);
```

See Also:
⟨undefined⟩ [Mix_RegisterEffect], page ⟨undefined⟩

4.6.5 Mix_SetPanning

`int` **Mix_SetPanning**`(int` *channel*`, Uint8` *left*`, Uint8` *right*`)`

channel Channel number to register this effect on.
Use **MIX_CHANNEL_POST** to process the postmix stream.

left Volume for the left channel, range is 0(silence) to 255(loud)

right Volume for the left channel, range is 0(silence) to 255(loud)

This effect will only work on stereo audio. Meaning you called `Mix_OpenAudio` with 2 channels (**MIX_DEFAULT_CHANNELS**). The easiest way to do true panning is to call `Mix_SetPanning(channel, left, 254 - left);` so that the total volume is correct, if you consider the maximum volume to be 127 per channel for center, or 254 max for left, this works, but about halves the effective volume.
This Function registers the effect for you, so don't try to `Mix_RegisterEffect` it yourself.
NOTE: Setting both *left* and *right* to 255 will unregister the effect from *channel*. You cannot unregister it any other way, unless you use `Mix_UnregisterAllEffects` on the *channel*.
NOTE: Using this function on a mono audio device will not register the effect, nor will it return an error status.

Returns: Zero on errors, such as bad channel, or if `Mix_RegisterEffect` failed.

```
// pan channel 1 halfway to the left
if(!Mix_SetPanning(1, 255, 127)) {
    printf("Mix_SetPanning: %s\n", Mix_GetError());
    // no panning, is it ok?
}
```

See Also:
⟨undefined⟩ [Mix_SetPosition], page ⟨undefined⟩, ⟨undefined⟩ [Mix_UnregisterAllEffects], page ⟨undefined⟩

4.6.6 Mix_SetDistance

`int` **Mix_SetDistance**`(int` *`channel`*`, Uint8` *`distance`*`)`

channel Channel number to register this effect on.
Use **MIX_CHANNEL_POST** to process the postmix stream.

distance Specify the distance from the listener, from 0(close/loud) to 255(far/quiet).

This effect simulates a simple attenuation of volume due to distance. The volume never quite reaches silence, even at max distance.
NOTE: Using a *distance* of 0 will cause the effect to unregister itself from *channel*. You cannot unregister it any other way, unless you use `Mix_UnregisterAllEffects` on the *channel*.

Returns: Zero on errors, such as an invalid channel, or if `Mix_RegisterEffect` failed.

```
// distance channel 1 to be farthest away
if(!Mix_SetDistance(1, 255)) {
    printf("Mix_SetDistance: %s\n", Mix_GetError());
    // no distance, is it ok?
}
```

See Also:
⟨undefined⟩ [Mix_SetPosition], page ⟨undefined⟩, ⟨undefined⟩ [Mix_UnregisterAllEffects], page ⟨undefined⟩

4.6.7 Mix_SetPosition

`int` **Mix_SetPosition**`(int` *`channel`*`, Sint16` *`angle`*`, Uint8` *`distance`*`)`

channel Channel number to register this effect on.
Use **MIX_CHANNEL_POST** to process the postmix stream.

angle Direction in relation to forward from 0 to 360 degrees. Larger angles will be reduced to this range using *`angles`* `%` `360`.
0 = directly in front.
90 = directly to the right.
180 = directly behind.
270 = directly to the left.
So you can see it goes clockwise starting at directly in front.
This ends up being similar in effect to `Mix_SetPanning`.

distance The distance from the listener, from 0(near/loud) to 255(far/quiet).
This is the same as the `Mix_SetDistance` effect.

This effect emulates a simple 3D audio effect. It's not all that realistic, but it can help improve some level of realism. By giving it the angle and distance from the camera's point of view, the effect pans and attenuates volumes. If you are looking for better positional audio, using **OpenAL** is suggested.
NOTE: Using *angle* and *distance* of 0, will cause the effect to unregister itself from *channel*. You cannot unregister it any other way, unless you use `Mix_UnregisterAllEffects` on the *channel*.

Returns: Zero on errors, such as an invalid channel, or if `Mix_RegisterEffect` failed.

```
// set channel 2 to be behind and right, and 100 units away
if(!Mix_SetPosition(2, 135, 100)) {
    printf("Mix_SetPosition: %s\n", Mix_GetError());
    // no position effect, is it ok?
}
```

See Also:
⟨undefined⟩ [Mix_SetPanning], page ⟨undefined⟩, ⟨undefined⟩ [Mix_SetDistance], page ⟨undefined⟩, ⟨undefined⟩ [Mix_UnregisterAllEffects], page ⟨undefined⟩

4.6.8 Mix_SetReverseStereo

`int` **Mix_SetReverseStereo**`(int` *`channel`*`, int` *`flip`*`)`

channel Channel number to register this effect on.
Use **MIX_CHANNEL_POST** to process the postmix stream.

flip Must be non-zero to work, means nothing to the effect processor itself.
set to zero to unregister the effect from *channel*.

Simple reverse stereo, swaps left and right channel sound.
NOTE: Using a *flip* of 0, will cause the effect to unregister itself from *channel*. You cannot unregister it any other way, unless you use `Mix_UnregisterAllEffects` on the *channel*.

Returns: Zero on errors, such as an invalid channel, or if `Mix_RegisterEffect` failed.

```
// set the total mixer output to be reverse stereo
if(!Mix_SetReverseStereo(MIX_CHANNEL_POST, 1)) {
    printf("Mix_SetReverseStereo: %s\n", Mix_GetError());
    // no reverse stereo, is it ok?
}
```

See Also:
⟨undefined⟩ [Mix_UnregisterAllEffects], page ⟨undefined⟩

5 Types

These types are defined and used by the SDL_mixer API.

5.1 Mix_Chunk

```
typedef struct {
    int allocated;
    Uint8 *abuf;
    Uint32 alen;
    Uint8 volume;     /* Per-sample volume, 0-128 */
} Mix_Chunk;
```

allocated a boolean indicating whether to free *abuf* when the chunk is freed.
0 if the memory was not allocated and thus not owned by this chunk.
1 if the memory was allocated and is thus owned by this chunk.

abuf Pointer to the sample data, which is in the output format and sample rate.

alen Length of *abuf* in bytes.

volume 0 = silent, 128 = max volume. This takes effect when mixing.

The internal format for an audio chunk. This stores the sample data, the length in bytes of that data, and the volume to use when mixing the sample.

See Also:
⟨undefined⟩ [Mix_VolumeChunk], page ⟨undefined⟩, ⟨undefined⟩ [Mix_GetChunk], page ⟨undefined⟩, ⟨undefined⟩ [Mix_LoadWAV], page ⟨undefined⟩, ⟨undefined⟩ [Mix_LoadWAV_RW], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FreeChunk], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Music], page ⟨undefined⟩

5.2 Mix_Music

```
typedef struct _Mix_Music Mix_Music;
```

This is an opaque data type used for Music data. This should always be used as a pointer. Who knows why it isn't a pointer in this typedef...

See Also:
⟨undefined⟩ [Mix_LoadMUS], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FreeMusic], page ⟨undefined⟩, ⟨undefined⟩ [Mix_Chunk], page ⟨undefined⟩

5.3 Mix_MusicType

```
typedef enum {
    MUS_NONE,
    MUS_CMD,
    MUS_WAV,
    MUS_MOD,
    MUS_MID,
    MUS_OGG,
    MUS_MP3
} Mix_MusicType;
```

Return values from `Mix_GetMusicType` are of these enumerated values.
If no music is playing then **MUS_NONE** is returned.
If music is playing via an external command then **MUS_CMD** is returned.
Otherwise they are self explanatory.

See Also:
⟨undefined⟩ [Mix_GetMusicType], page ⟨undefined⟩

5.4 Mix_Fading

```
typedef enum {
    MIX_NO_FADING,
    MIX_FADING_OUT,
    MIX_FADING_IN
} Mix_Fading;
```

Return values from `Mix_FadingMusic` and `Mix_FadingChannel` are of these enumerated values. If no fading is taking place on the queried channel or music, then **MIX_NO_FADING** is returned. Otherwise they are self explanatory.

See Also:
⟨undefined⟩ [Mix_FadingChannel], page ⟨undefined⟩, ⟨undefined⟩ [Mix_FadingMusic], page ⟨undefined⟩

5.5 Mix_EffectFunc_t

```
typedef void (*Mix_EffectFunc_t)(int chan, void *stream, int len,
                                 void *udata);
```

chan The channel number that this effect is effecting now.
MIX_CHANNEL_POST is passed in for post processing effects over the final mix.

stream The buffer containing the current sample to process.

len The length of *stream* in bytes.

udata User data pointer that was passed in to `Mix_RegisterEffect` when registering this effect processor function.

This is the prototype for effect processing functions. These functions are used to apply effects processing on a sample chunk. As a channel plays a sample, the registered effect functions are called. Each effect would then read and perhaps alter the *len* bytes of *stream*. It may also be advantageous to keep the effect state in the *udata*, with would be setup when registering the effect function on a channel.

See Also:
⟨undefined⟩ [Mix_RegisterEffect], page ⟨undefined⟩ ⟨undefined⟩ [Mix_UnregisterEffect], page ⟨undefined⟩

5.6 Mix_EffectDone_t

```
typedef void (*Mix_EffectDone_t)(int chan, void *udata);
```

chan The channel number that this effect is effecting now.
MIX_CHANNEL_POST is passed in for post processing effects over the final mix.

udata User data pointer that was passed in to `Mix_RegisterEffect` when registering this effect processor function.

This is the prototype for effect processing functions. This is called when a channel has finished playing, or halted, or is deallocated. This is also called when a processor is unregistered while processing is active. At that time the effects processing function may want to reset some internal variables or free some memory. It should free memory at least, because the processor could be freed after this call.

See Also:
⟨undefined⟩ [Mix_RegisterEffect], page ⟨undefined⟩ ⟨undefined⟩ [Mix_UnregisterEffect], page ⟨undefined⟩

6 Defines

MIX_MAJOR_VERSION

`1`

SDL_mixer library major number at compilation time

MIX_MINOR_VERSION

`2`

SDL_mixer library minor number at compilation time

MIX_PATCHLEVEL

`4`

SDL_mixer library patch level at compilation time

MIX_CHANNELS

`8`

The default mixer has this many simultaneous mixing channels after the first call to `Mix_OpenAudio`.

MIX_DEFAULT_FREQUENCY

`22050`

Good default sample rate in Hz (samples per second) for PC sound cards.

MIX_DEFAULT_FORMAT

`AUDIO_S16SYS`

The suggested default is signed 16bit samples in host byte order.

MIX_DEFAULT_CHANNELS

`2`

Stereo sound is a good default.

MIX_MAX_VOLUME

`128`

Maximum value for any volume setting.
This is currently the same as **SDL_MIX_MAXVOLUME**.

MIX_CHANNEL_POST

`-2`

This is the channel number used for post processing effects.

MIX_EFFECTSMAXSPEED

`"MIX_EFFECTSMAXSPEED"`

A convience definition for the string name of the environment variable to define when you desire the internal effects to sacrifice quality and/or RAM for speed. The environment variable must be set (else nonexisting) before `Mix_OpenAudio` is called for the setting to take effect.

7 Glossary

Byte Order
Also known as *Big-Endian*. Which means the most significant byte comes first in storage. Sparc and Motorola 68k based chips are MSB ordered.
(SDL defines this as **SDL_BYTEORDER==SDL_BIG_ENDIAN**)

Little-Endian(LSB) is stored in the opposite order, with the least significant byte first in memory. Intel and AMD are two LSB machines.
(SDL defines this as **SDL_BYTEORDER==SDL_LIL_ENDIAN**)

Index

(Index is nonexistent)