AudioMixer v1.0

Programmer's Guide

Table of Contents

1 Introduction....1
2 Using the AudioMixer API....1
  2.1 Checking API version....1
  2.2 Creating a new client....2
  2.3 Getting and putting buffers....2
  2.4 Destroying clients....3
  2.5 Advanced features....3
    2.5.1 Buffer management....3
    2.5.2 Timestamp management....4
3 AudioMixer API Reference....6
  3.1 Defines....6
    3.1.1 AUDMIX_VERSION_INT....6
    3.1.2 AUDMIX_VERSION_STR....6
  3.2 Types, Structures....6
    3.2.1 audmixClient_t....6
    3.2.2 audmixBufferFormat_t....7
    3.2.3 audmixInfoBuffer_t....8
  3.3 Functions....9
    3.3.1 AudMixGetInformation();....9
    3.3.2 AudMixStartDaemonThread();....10
    3.3.3 AudMixStopDaemonThread();....11
    3.3.4 AudMixCreateClient();....11
    3.3.5 AudMixDestroyClient();....12
    3.3.6 AudMixResizeBuffer();....12
    3.3.7 AudMixAddNewBuffer();....13
    3.3.8 AudMixDestroyBuffer();....14
    3.3.9 AudMixGetEmptyBuffer();....14
    3.3.10 AudMixPutFullBuffer();....15
    3.3.11 AudMixPutEmptyBuffer();....16
    3.3.12 AudMixGetNumOfBuffers();....17
    3.3.13 AudMixWaitForTimestampUpdate();....17
    3.3.14 AudMixGetTimestampInfo();....18

Last documentation update: 2006.05.23.

1 Introduction

AudioMixer is a software library which is meant to be able to mix multiple audio streams from different client softwares running on the computer. It was basically born out of the need of a precise audio mixer library for the Triton multimedia player project, with the capability of reporting the current exact position of every stream.

The AudioMixer library consists of two main parts: the daemon process which deals with the sound hardware and does the real mixing and sound format conversion (if needed), and the client processes feeding the daemon process with audio data to be played back.

The library uses the Triton Porting Layer (TPL) for the high resolution timing. The binary form of the library consists of a daemon process, which is a running AudMixD.exe application, and an AudMixer.dll file which can be linked and used by the client processes to be able to use the AudioMixer API.

2 Using the AudioMixer API

Assuming that the AudioMixer daemon is running (which should be the case normally), the client code can use the AudioMixer API through the AudMixer.dll file. The client application has to include the AudMixer.h file to get the definitions and prototypes of the AudioMixer API, and tell the linker to link to the AudMixer.lib file to resolve the references to these functions.

2.1 Checking API version

First of all, every application using the AudioMixer API should check if the AudioMixer daemon is available, running, and is at the right version level. For this, the `AudMixGetInformation()` API can be used:

```
#include <stdlib.h>
#include <stdio.h>
#include "tpl.h"
#include "AudMixer.h"

int main(int argc, char *argv[])
{
  audmixInfoBuffer_t AudMixInfo;
  int rc;

  rc = AudMixGetInformation(&AudMixInfo, sizeof(AudMixInfo));
  if (!rc)
  {
    printf("* Error querying Audio Mixer Information!\n");
    return 1;
  }

  printf("Using mixer: %s\n", AudMixInfo.pchVersion);

  if (AudMixInfo.iVersion<AUDMIX_VERSION_INT)
```

```
  {
    printf("* Audio Mixer is too old! Wanted v%d.%02d, found v%d.%02d\n",
           AUDMIX_VERSION_INT/10, AUDMIX_VERSION_INT%10,
           AudMixInfo.iVersion/10, AudMixInfo.iVersion%10);
    return 1;
  }

  if (!AudMixInfo.bIsDaemonRunning)
  {
    printf("* Audio Mixer Daemon is not running!\n");
    return 1;
  }

  return 0;
}
```

2.2 Creating a new client

Once the API version check is done, the application can register into the daemon as a new client. This can be done by using the `AudMixCreateClient()` function, telling the required initial number of audio buffers, and the size of each of these buffers. If this function call succeeds (returns a non-NULL value as the client handle), then the audio daemon is successfully registered us as a new audio stream source, and it's ready to accept buffers from us filled with audio data to be played back. It worths noting that one process or thread is not limited to create only one AudioMixer client, it can register multiple audio stream sources.

2.3 Getting and putting buffers

When a new client is registered into the daemon, the given number of buffers is allocated in the high shared memory area with the given buffer size for each. These buffers are empty by default, and they are waiting at the the daemon to be taken out by the client.

The `AudMixGetEmptyBuffer()` function can be used to get one of the empty buffers of the given client. It is capable of waiting for an empty buffer to be available (with a given timeout for the waiting) in case there is no empty buffer available at the time of the call.

So, once an empty buffer is got with the `AudMixGetEmptyBuffer()` API, the code can fill this buffer with audio data, and give the buffer back to the AudioMixer daemon as a full buffer to be played back using the `AudMixPutFullBuffer()` API. There is also a possibility of giving the buffer back as an unused (empty) one in case the buffer was not used, using the `AudMixPutEmptyBuffer()` API.

A normal audio playback can be reached by having a loop of getting empty buffers, filling them with audio data, and putting back full buffers:

```
  while(1)
  {
    rc = AudMixGetEmptyBuffer(hAudMixClient,
```

```
                              &pBuffer,
                              &uiBufferSize, 250); /* 250 msec timeout */
  if (rc)
  {
    printf("* Got an empty buffer 0x%p\n", pBuffer);

    /* ... TODO: Fill pBuffer with audio data ... */

    printf("* Sending a full buffer (size is %d)\n", uiBufferSize);

    rc = AudMixPutFullBuffer(hAudMixClient,
                             pBuffer,
                             uiBufferSize,
                             ullPlaybackPos, 1000,
                             &fmtBufferFormat);
    printf("* Sent a full buffer (rc is %d), pos %d\n", rc, (int) ullPlaybackPos);
  }
}
```

2.4 Destroying clients

When the application stops using the AudioMixer, the client handles returned by the `AudMixCreateClient()` function has to be destroyed by calling the `AudMixDestroyClient()` function. Although the AudMixer.dll is built that way that it destroys registered clients at process die time in case the programmer forgets to do so (or the application crashes), it's always a good idea and good practice to clean up everything.

2.5 Advanced features

There are some more AudioMixer functions to deal with the buffers and the timestamps. They might not be needed for the average usage, but they come handy when the audio stream suddenly needs extra buffers, or when some kind of synchronization to the audio stream is needed.

2.5.1 Buffer management

In addition to getting and putting buffers from and to the AudioMixer daemon, it's also possible to resize a given buffer using the `AudMixResizeBuffer()` function. It can only be called for buffers previously got from the AudioMixer daemon with the `AudMixGetEmptyBuffer()` call, and it only resizes that particular buffer.

It may also happen that the initial number of buffers becomes too small, and the application will need additional buffers. This can be solved by calling the `AudMixAddNewBuffer()` function. That call will create a new empty buffer at the daemon side, which can then be got by calling the `AudMixGetEmptyBuffer()` function.

Once it's possible to increase the number of buffers of a client, it would be silly if it wouldn't be possible to decrease it. The `AudMixDestroyBuffer()` function can be used to destroy an empty buffer, got

from the AudioMixer daemon by the `AudMixGetEmptyBuffer()` API.

The current state of the buffers for a given AudioMixer client can be got by using the `AudMixGetNumOfBuffers()` API. This call will return the total number of buffers belonging to a given client, and will also return the number of buffers at the daemon side.

2.5.2 Timestamp management

Precise synchronization to an audio stream requires good timestamp management. AudioMixer is capable of handling timestamped audio buffers, and it's able to report the current playback position based on these timestamps.

When the `AudMixPutFullBuffer()` function is used to put a given chunk of audio data into the AudioMixer to be played back, it's possible to attach two 64bits (unsigned long long typed) values to every buffer. The first one is called `ullTimeStamp`, the second one is `ullTimeStampIncPerSec`.

The `ullTimeStamp` is the timestamp that belongs to the very first audio sample of the buffer. It means that when the AudioMixer reaches the first audio sample of this buffer in the playback, it will report this timestamp as the current playbck position.

The `ullTimeStampIncPerSec` tells the system the unit in which the `ullTimeStamp` is given. For example, a value of `ullTimeStampIncPerSec` of 1000 means that the timestamp should increase 1000 units in one second, meaning that the timestamps are basically telling the position in milliseconds.

Using these two values, the AudioMixer can always tell the exact playback position of the audio stream. For example, when the playback reaches the half of a given buffer, AudioMixer will calculate how many time has been elapsed since the playback of the very first audio sample, and increases the `ullTimeStamp` value of the buffer according to that value, and reports this result as the current position.
Let's say we have a buffer of 8192 bytes, containing 16bits Stereo audio samples with the samplerate of 44100 Hz, with `ullTimeStamp` value of 2400 and `ullTimeStampIncPerSec` value of 1000. The 8192 bytes audio data with the given audio format (4 bytes per sample, 44.1KHz) means that the length of the audio buffer is 0.046 seconds (46 milliseconds). Being at the half of the buffer means that half of the buffer length was already played back, so 0.023 seconds were already played back, so the current timestamp will be 2423 (`ullTimeStamp + 0.023*ullTimeStampIncPerSec`).

Of course, the AudioMixer does not do the mixing in an continuous way, it mixes audio data from the clients in “bursts”, or in “chunks”. It means that it usually mixes some audio data from the client's audio stream, then it waits for that small audio chunk to be played back, then mixes some more, and waits again, and so on. This results in the effect that the current timestamp for the audio clients will “jump” and only change when a given time has elapsed. This time depends on the configuration of the AudioMixer daemon, but it's usually around 23 milliseconds. A timestamp precision of 23 milliseconds is not a very good one for audio and video synchronization, so an additional help is used to make it even more precise.

As the AudioMixer itself knows the length of its own internal audio buffers, it knows how long they are, so it can tell the time between these mixing “bursts”. It can also tell when the previous mixing (with the

given timestamp result) was done in the system, so it can report that this particular timestamp was reached at this global system time, and it won't change (I won't be mixing any more) for this much time. Knowing the current global system time, it's very easy to increase the timestamp according to the difference of the current global system time and the global system time when the timestamp was reached, this way getting a very very accurate position information.

The Triton Porting Layer (TPL) is used for this global system time. The TPL has support for high resolution timer, which is a free-running 64bits counter, and another 64bits value telling how many units this counter increases in one second. This free-running counter is used as the global system time. Please check the `tpl_hrtimerGetTime()` and the `tpl_hrtimerGetOneSecTime()` functions in TPL for more information about the TPL high resolution timer.

The `AudMixGetTimestampInfo()` API is capable of reporting the values belonging to a given client, discussed above: timestamp of audio stream, the global system time when that timestamp was reached, and the time (in global system time units) for how long this timestamp will stay the same (so the time that is between the timestamp updates).

The `AudMixWaitForTimestampUpdate()` API can be used to wait for a timestamp update event for the given client. Once this event occures, the AudMixGetTimestampInfo() will (hopefully) return a different timestamp, because it has processed/mixed an additional chunk of the client's audio stream.

3 AudioMixer API Reference

3.1 Defines

3.1.1 AUDMIX_VERSION_INT

```
#define AUDMIX_VERSION_INT     100
```

This define contains the version number of the AudioMixer library for which the current header file belongs to. It's a combination of both the major and minor version number, constructed by multiplying the major version number by 100 and adding the minor version number (e.g.: version 1.00 will be defined to the value 100).

This define can be used to check at runtime if the AudioMixer library that's present on the system has the correct version number (see the `audmixInfoBuffer_t` type and the `AudMixGetInformation()` function).

3.1.2 AUDMIX_VERSION_STR

```
#define AUDMIX_VERSION_STR   "1.00"
```

This define contains the version number of the AudioMixer library for which the current header file belongs to, in a textual form. It may contain additional information compared to `AUDMIX_VERSION_INT`, like sub-subversion info, or different other build-related things.

This define can be used to tell the user the exact version of AudioMixer which is (minimally or exclusively) required for the application.

3.2 Types, Structures

3.2.1 audmixClient_t

```
typedef struct audmixClient_s audmixClient_t, *audmixClient_p;
```

Description:

This type is a private, internal type of the AudioMixer library, describing the given AudioMixer client and its internal state. Almost all of the AudioMixer API functions need a client handle of this type.

3.2.2 audmixBufferFormat_t

```
typedef struct audmixBufferFormat_s
{
  unsigned int  uiSampleRate;
  unsigned int  uiChannels;
  unsigned char uchBits;
  unsigned char uchSigned;
  unsigned char uchDoReversePlay;
  unsigned char uchReserved1;
  unsigned int  uiVolume;     /* 0..256, 0 is Mute, 256 is full volume */

} audmixBufferFormat_t, *audmixBufferFormat_p;
```

Fields:

`uiSampleRate:`
The audio sample rate in Hz.
For example: 44100

`uiChannels:`
The number of audio channels per audio sample.
For example: 1 (for mono), 2 (for stereo, L-R), etc...

`uchBits:`
The number of bits per one channel of an audio sample.
For example: 8, 16

`uchSigned:`
Tells if the audio samples are signed or unsigned. A zero value means that the audio samples are unsigned, a non-zero value means that the audio samples are signed.
Signed audio samples are so that a 0 value for an audio sample means that it's at the center, while a 0 value for an unsigned audio sample means that it's at the bottom limit of its dynamic range.
For example: 1 (for signed), 0 (for unsigned)

`uchDoReversePlay:`
Tells if the audio buffer described by this structure should be played back starting from its last audio sample, advancing towards its first audio sample, or in a normal way, starting from the first audio sample and advancing towards its last audio sample.
A value of zero means that the audio buffer will be played back in a normal way, while any non-zero value will have the effect of reverse playback.
For example: 1 (for reverse playback), 0 (for normal playback)
Note: It's not yet implemented!

`uiVolume:`
Tells the audio volume of this stream. A value of zero will mean that this audio stream is muted, so the audio data sent to the system will not be hearable at all. A value of 256 will mean that this audio stream will be mixed to the other streams with full volume. Any other value between 0 and 256 is valid, the closer to 0 the lower the volume will be.
For example: 0 (for mute), 128 (for half-volume), 256 (for full volume)

Description:

This structure describes the format of a given audio buffer, and also contains some hints on how to play it back. Every time a full audio buffer is sent to the system, an `audmixBufferFormat_t` structure must also be given with the audio data to describe it.

3.2.3 audmixInfoBuffer_t

```
typedef struct audmixInfoBuffer_s
{
  int   iVersion;
  char *pchVersion;

  int   bIsDaemonRunning;

  unsigned int  uiPrimarySampleRate;
  unsigned int  uiPrimaryChannels;
  unsigned char uchPrimaryBits;
  unsigned char uchPrimarySigned;

  unsigned int  uiPrimaryNumBuffers;
  unsigned int  uiPrimaryBufSize;

  /* Might be extended later */
} audmixInfoBuffer_t, *audmixInfoBuffer_p;
```

Fields:

`iVersion:`
Version number of the AudioMixer library available in the system. It's in the same format as the `AUDMIX_VERSION_INT` define, so it can be compared to that value to decide if the AudioMixer library that is available on the current system is usable with the application or not.

`pchVersion:`
Version number of the AudioMixer library available in the system, in a textual form. It can be used in case of version mismatch error, to tell the user what version was found on the system instead of the required version.

`bIsDaemonRunning:`
It's a boolean value (containing 0 or 1), telling if the AudioMixer Daemon is currently running or not. The current implementation requires a Daemon process to be run for the mixing to be done. If the Daemon is not running, the AudioMixer library will not be usable, and most of the API calls will return with error.

`uiPrimarySampleRate:`
Valid only if the Daemon is running (see `bIsDaemonRunning` field). Part of the audio format description of the primary buffers used by the Daemon. This is the format in which the mixed audio data is finally sent to the sound card.

`uiPrimaryChannels:`
Valid only if the Daemon is running (see `bIsDaemonRunning` field). Part of the audio format description of the primary buffers used by the Daemon. This is the format in which the mixed audio data is finally sent to the sound card.

`uchPrimaryBits:`
Valid only if the Daemon is running (see `bIsDaemonRunning` field). Part of the audio format description of the primary buffers used by the Daemon. This is the format in which the mixed audio data is finally sent to the sound card.

`uchPrimarySigned:`
Valid only if the Daemon is running (see `bIsDaemonRunning` field). Part of the audio format description of the primary buffers used by the Daemon. This is the format in which the mixed audio data is finally sent to the sound card.

`uiPrimaryNumBuffers:`
Valid only if the Daemon is running (see `bIsDaemonRunning` field). Tells the number of primary buffers circulated in the audio hardware by the audio mixer Daemon.

`uiPrimaryBufSize:`
Valid only if the Daemon is running (see `bIsDaemonRunning` field). Tells the size of the primary buffers circulated in the audio hardware by the audio mixer Daemon, in bytes.

Description:

This structure is filled by the `AudMixGetInformation()` API, giving general information about the AudioMixer library in the system.

3.3 Functions

3.3.1 AudMixGetInformation();

```
int AudMixGetInformation(audmixInfoBuffer_p pInfoBuffer,
                         unsigned int uiInfoBufferSize);
```

Parameters:

`pInfoBuffer:`
Pointer to an audmixInfoBuffer_t structure, which will be filled by the function. See the description of the structure format itself for more information.

`uiInfoBufferSize:`
The number of bytes to be put into the memory area pointed by the pInfoBuffer parameter. The memory area will be filled with zeros first, and then the known fields of the structure will be filled.

Returns:

0 – in case of invalid parameter(s)
1 – in any other case (success)

Description:

This function is to be called before any other AudioMixer API functions. It can be used to get general information about the AudioMixer library installed in the current system. The function will fill the `audmixInfoBuffer_t` structure for which the address is passed in the `pInfoBuffer` parameter. To make it usable with future AudioMixer library versions, the `uiInfoBufferSize` parameter must contain the size of the `audmixInfoBuffer_t` structure that is to be filled, so if this structure will grow in later versions, this function will still not overwrite memory for clients built with older versions.

3.3.2 AudMixStartDaemonThread();

```
int AudMixStartDaemonThread(unsigned int uiNumBuffers,
                            unsigned int uiBufSize);
```

Parameters:

`uiNumBuffers:`
Number of primary buffers to use for the mixing. A minimum of 2 must be used.

`uiBufSize:`
The size of one primary buffer to use for the mixing.

Returns:
0 – in case of invalid parameter(s),
if the Daemon thread is already present in the system,
if the Daemon thread could not be created for some reason,
if the Daemon thread could not be configured for some reason.
1 – in any other case (success)

Description:

This function starts a new thread in the calling process, which serve the AudioMixer clients, and will do the mixing of audio data coming from the clients.

Only one AudioMixer Daemon can be on the same system at the same time. This function should be used only by the AudioMixer Daemon executable, which should be started at the system startup time, so there are chances that this function will always return failure for the average coder.

Please note that the format of the primary buffer is not changeable in the current implementation, but once it will be configurable, this API is likely to change to have more parameters.

3.3.3 AudMixStopDaemonThread();

```
int AudMixStopDaemonThread();
```

Parameters:

None.

Returns:

0 – if the Daemon thread is not yet running,
if the Daemon thread is not in the calling process,
if the Daemon thread could not shut down gracefully.
1 – in any other case (success)

Description:

This function is used to notify the AudioMixer Daemon thread to free all the resources it allocated, disconnect all the clients, and destroy itself.

It can only be called from the same process that started the Daemon thread with the `AudMixStartDaemonThread()` API call, otherwise the API will return failure.

This function should only be used by the AudioMixer Daemon application, which is started at system startup time. That application calls this function when it's terminating.

3.3.4 AudMixCreateClient();

```
audmixClient_p AudMixCreateClient(unsigned int uiNumBuffers,
                                  unsigned int uiBufSize);
```

Parameters:

`uiNumBuffers:`
The initial number of audio buffers to be created for this client. A minimum of 2 buffers has to be created.

`uiBufSize:`
The size (in bytes) of each of the audio buffers to be created for this client initially. The minimum acceptable size is 512 bytes per buffer.

Returns:

NULL – if the Daemon thread is not yet running,
if any of the parameters are invalid,
if any of the required resources (shared memory, semaphores) could not be created.
Not-NULL – in any other case (success),
containing the new client handle to be used for subsequent AudioMixer API calls.

Description:

New AudioMixer clients can be created by using this function. The returned value (if it's not NULL) can be used as the client-handle for the AudioMixer API calls.

It's allowed for one process or thread to create more than one clients.

The number and size of the audio buffers for this client can be modified later with AudioMixer API calls like `AudMixAddNewBuffer()`, `AudMixDestroyBuffer()`, `AudMixResizeBuffer()`.

3.3.5 AudMixDestroyClient();

```
int AudMixDestroyClient(audmixClient_p hClient);
```

Parameters:

`hClient:`
The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

Returns:
0 – if the Daemon thread is not yet running,
if the parameter is invalid,
if the given client was not created by the calling process.
1 – in any other case (success)

Description:

All the audio clients created by the `AudMixCreateClient()` API must be destroyed using this API to free all the resources allocated for that client.

The function will disconnect the client from the Daemon, and free all the resources (shared memory, semaphores) allocated for the client.

The current AudioMixer implementation will automatically clean up clients that are still not destroyed at the time of process termination (and belong to that given client, of course), but still, it's a good practice to clean up everything created.

3.3.6 AudMixResizeBuffer();

```
int AudMixResizeBuffer(audmixClient_p hClient,
                       void **ppBuffer,
                       unsigned int uiNewBufSize);
```

Parameters:

`hClient:`
The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

`ppBuffer:`
The address of a pointer variable containing the audio buffer which needs to be resized. The value stored in the pointer variable will be updated after the call, so in other words, the address of the new (resized) buffer will be different than the address of the old buffer!

`uiNewBufSize:`
The new required size (in bytes) of the audio buffer.

Returns:
0 – if the Daemon thread is not yet running,
if any of the parameters are invalid,
if the buffer is still inside the Daemon (was not taken out by `AudMixGetEmptyBuffer()`),
if the given client was not created by the calling process.
1 – in any other case (success)

Description:

The size of the audio buffers can be changed on the fly. Only that process can change the size which created the client and the buffers.

If the resize is successful, the resized buffer will have a new address, so the pointer (for which the address was sent to the API) will now contain a new value. The old value is not valid anymore.

Only that buffer can be resized which is not inside the Daemon, so which was got by the `AudMixGetEmptyBuffer()` API.

3.3.7 AudMixAddNewBuffer();

```
int AudMixAddNewBuffer(audmixClient_p hClient,
                       unsigned int uiBufSize);
```

Parameters:

`hClient:`
The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

`uiBufSize:`
The size of the new audio buffer (in bytes) to be created and added into the pool of audio buffers available to this client.

Returns:
0 – if the Daemon thread is not yet running,
if any of the parameters are invalid,
if the given client was not created by the calling process.

1 – in any other case (success)

Description:

The number of the audio buffers for a client can be changed on the fly. One way to change it is to add a new buffer. This function will add a new buffer with the given size into the client's pool of buffers. The new buffer will be empty, and ready to be got by the `AudMixGetEmptyBuffer()` API.

3.3.8 AudMixDestroyBuffer();

```
int AudMixDestroyBuffer(audmixClient_p hClient,
                        void *pBuffer);
```

Parameters:

`hClient:`
The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

`pBuffer:`
The address of the buffer which has to be destroyed.

Returns:
0 – if the Daemon thread is not yet running,
if any of the parameters are invalid,
if the buffer is still inside the Daemon (was not taken out by `AudMixGetEmptyBuffer()`),
if the given client was not created by the calling process.
1 – in any other case (success)

Description:

The number of the audio buffers for a client can be changed on the fly. One way to change it is to destroy an old buffer. This function will destroy an old buffer of the given client.

Only that buffer can be destroyed which is not inside the Daemon, so which was got by the `AudMixGetEmptyBuffer()` API.

3.3.9 AudMixGetEmptyBuffer();

```
int AudMixGetEmptyBuffer(audmixClient_p hClient,
                         void **ppBuffer,
                         unsigned int *puiBufferSize,
                         int iTimeOut);
```

Parameters:

`hClient:`

The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

`ppBuffer:`
The address of a pointer which will contain the address of the empty audio buffer after successful run of the function.

`puiBufferSize:`
Address of an unsigned int variable which will contain the maximum number of bytes that can be put into the buffer.

`iTimeOut:`
The number of milliseconds to wait for a new Empty buffer to become available. A value of -1 means infinite waiting.

Returns:

0 – if the Daemon thread is not yet running,
if any of the parameters are invalid,
if the given client was not created by the calling process,
if no buffer became empty in the given time, so timeout occurred.
1 – in any other case (success)

Description:

When the AudioMixer Daemon does the mixing, it processes audio bytes from the full buffers that are sent into the Daemon. Once every byte of a full buffer is consumed, that buffer becomes empty, and that can be taken out of the Daemon by the client for which it belongs to. The client can fill that empty buffer then (or resize, or destroy...) and send back to the Daemon for further mixing.

This function can be used to wait for an empty buffer to become available, and take that buffer out of the Daemon. The function will return the address and size of the buffer.

3.3.10 AudMixPutFullBuffer();

```
int AudMixPutFullBuffer(audmixClient_p hClient,
                        void *pBuffer,
                        unsigned int uiDataInside,
                        unsigned long long ullTimeStamp,
                        unsigned long long ullTimeStampIncPerSec,
                        audmixBufferFormat_p pfmtFormat);
```

Parameters:

`hClient:`
The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

`pBuffer:`
The buffer to be given back to AudioMixer Daemon for mixing.

`uiDataInside:`
The number of bytes actually put into the buffer. Please note that it's not the size of the buffer, but the number of bytes that are actually put in the buffer (can be a smalelr value than the size of the buffer).

`ullTimeStamp:`
A 64bits unsigned value. It's the timestamp of the very first sample of the buffer.

`ullTimeStampIncPerSec:`
A 64bits unsigned value. It tells the system how much it should increase on the timestamp given above in once second.

`pfmtFormat:`
Pointer to an `audmixBufferFormat_t` structure, describing the format of the audio data put into the audio buffer. Please see the description of the `audmixBufferFormat_t` structure for more information.

Returns:
0 – if the Daemon thread is not yet running,
if any of the parameters are invalid,
if the given client was not created by the calling process,
if the given audio format is not valid.
1 – in any other case (success)

Description:

When the application gets empty audio buffers from the AudioMixer, it can fill it with audio data and send the full buffer back to the Daemon for playback. This function can be used to send back the full buffers.

The caller can give timestamp information optionally. If the `ullTimeStampIncPerSec` is not zero, then timestamp information will be calculated by the AudioMixer Daemon as it does the mixing.

3.3.11 AudMixPutEmptyBuffer();

```
int AudMixPutEmptyBuffer(audmixClient_p hClient,
                         void *pBuffer);
```

Parameters:

`hClient:`
The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

`pBuffer:`
The buffer to be given back to AudioMixer Daemon as an empty one.

Returns:
0 – if the Daemon thread is not yet running,

if any of the parameters are invalid,
if the given client was not created by the calling process
1 – in any other case (success)

Description:

When the application gets empty audio buffers from the AudioMixer, it can fill it with audio data and send the full buffer back to the Daemon for playback. However, it's also possible that the application decides that it has no more audio data to play back, so it can also give back the buffer as an empty one. This function can be used to send back the empty buffers.

3.3.12 AudMixGetNumOfBuffers();

```
int AudMixGetNumOfBuffers(audmixClient_p hClient,
                          unsigned int *puiNumAllBufs,
                          unsigned int *puiNumInSystem);
```

Parameters:

`hClient:`
The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

`puiNumAllBufs:`
Pointer to an unsigned int variable, in which the number of all the buffers allocated for this client is returned.

`puiNumInSystem:`
Pointer to an unsigned int variable, in which the number of all the buffers for this client is returned that are at the Daemon. This number is always less or equal to the value returned in `puiNumAllBufs`.

Returns:
0 – if the Daemon thread is not yet running,
if any of the parameters are invalid,
if the given client was not created by the calling process
1 – in any other case (success)

Description:

The application can dynamically modify the number of audio buffers of a given AudioMixer client by allocating, resizing and destroying audio buffers. This function can be used to get the current status of a given client. The function returns the sum number of buffers that are allocated for this client, and the number of buffers that can still be got by `AudMixGetEmptyBuffer()` API.

3.3.13 AudMixWaitForTimestampUpdate();

```
int AudMixWaitForTimestampUpdate(int iTimeOut);
```

Parameters:

`iTimeOut:`
The maximum time (in milliseconds) to wait for a timestamp update event.

Returns:
0 – if the Daemon thread is not yet running,
if no timestamp update event has occurred in the given time (timeout)
1 – in any other case (success)

Description:

The AudioMixer Daemon always does the mixing of audio data got from the current AudioMixer clients. The mixing is done in “bursts”, jumping in the audio streams of the clients chunk by chunk. Once the AudioMixer Daemon fills one of its internal audio buffers by mixing audio data from the clients, it also updates the timestamp information for the clients. Once this timestamp information is updated then a timestamp update event occurs.
This function can be used to wait for such an event. The application then can query the current (updated) timestamp information for the AudioMixer client by using the `AudMixGetTimestampInfo()` API.

3.3.14 AudMixGetTimestampInfo();

```
int AudMixGetTimestampInfo(audmixClient_p hClient,
                           unsigned long long *pullTimeStamp,
                           tplTime_t *ptimTimeStampTime,
                           tplTime_t *ptimTimeStampUpdateInterval);
```

Parameters:

`hClient:`
The AudioMixer Client handle returned by the `AudMixCreateClient()` API.

`pullTimeStamp:`
Pointer to an unsigned long long variable, in which the last calculated timestamp information for the given client will be returned.

`ptimTimeStampTime:`
Pointer to an `tplTime_t` variable, in which the system time will be returned, when the given timestamp was reached. See the Triton Porting Layer for more information about `tplTime_t`!

`ptimTimeStampUpdateInterval:`
Pointer to an `tplTime_t` variable, in which the it's returned how much time is needed (how much the system time counter increases) for a timestamp update. In other words, it returns the length of the AudioMixer's internal buffers, in system time. See the Triton Porting Layer for more information about `tplTime_t`!

Returns:

0 – if the Daemon thread is not yet running,
if any of the parameters are invalid,
if the given client was not created by the calling process

1 – in any other case (success)

Description:

In order to achieve very precise synchronization, the clients has to know the current position of the playback. This function can be used to query the current timestamp of the given AudioMixer client (provided that the AudioMixer got valid timestamp information at `AudMixPutFullBuffer()`), and this timestamp can be even more refined using the additional information returned by this function. Using the system time API from the Triton Porting Layer (see `tpl_hrtimerGetTime()` and `tpl_hrtimerGetOneSecTime()` functions in there) and these additional information, the exact playback position can be easily calculated.