Headspace Audio Engine: Host Interface API

Document Version 1.00
Modified 4/27/98
Reflects HAE SDK Release 1.0

Context

The Headspace Audio Engine (HAE) is a machine independent, audio device independent, software-only runtime library that offers music synthesis and digital audio playback services for entertainment, media production, and scientific sonification applications. HAE's Host Interface API is the C language interface that makes HAE's machine independence possible by placing an abstraction layer at the boundary between HAE and its host system (OS services and audio hardware).

Synopsis

This document is intended to teach systems programmers how to use the HAE Host Interface API to adapt HAE to operate on a new host system. The Host Interface API specifies the interfaces for a set of functions, most of which are go-between routines that you must write; except for one callable HAE function, this is not an ordinary API of service functions that you call. Instead, HAE calls **your** functions when it needs to communicate with the host OS and hardware.

HAE has its own models of processes and buffered digital audio output streams, as do all host systems, and so in writing these functions you may occasionally find yourself in the position of having to mediate between the two models.

Related Documentation

For information on HAE's general capabilities and its main user services API, please refer to the **HAE Client API** document.

For information on using HAE to create RMF files, please refer to the **HAE Exporter API** document.

For musicians' and sound artists' information on preparing music, dialog, and sound effects content for HAE playback, please refer to:

http://www.headspace.com/beatnik/doc

Contents

An Introduction to the HAE Host Interface API

HAE’s Host Interface API is the C language interface that makes HAE’s machine independence possible by placing an abstraction layer at the boundary between HAE and the host system (OS services and audio hardware).

HAE’s other APIs generally define suites of facilities that HAE furnishes, but the Host Interface API is different. This API can be thought of as defining the set of requirements that HAE imposes on any environment in which it is asked to operate. For the most part, the Host Interface API defines a set of functions that you must write, not functions that you call, and HAE calls these functions when it needs to communicate with the OS and the hardware. To HAE, these functions you write appear as `extern "C"` functions.

In writing these functions, you may occasionally find yourself in the position of having to mediate between the potentially different models of processes and digital audio output that HAE and your machine hold. For example, HAE uses a thread model, so the Host Interface implementation for Mac OS (which has interrupts but no threads) had to call HAE’s audio output generation function at interrupt time.

But don’t be daunted; HAE is written to make porting as simple as possible. As of early 1998, the Host Interface API has been used to adapt HAE to the following hosts: Mac OS, Windows 3.1, Windows 95/NT, Solaris, Java virtual machines, WebTV, Netscape plug-ins, and Active X, as well as a number of other, proprietary systems and new audio output devices working under the above host operating systems. These adaptations have been implemented both by Headspace software engineers and by the software engineers of HAE licensee developers.

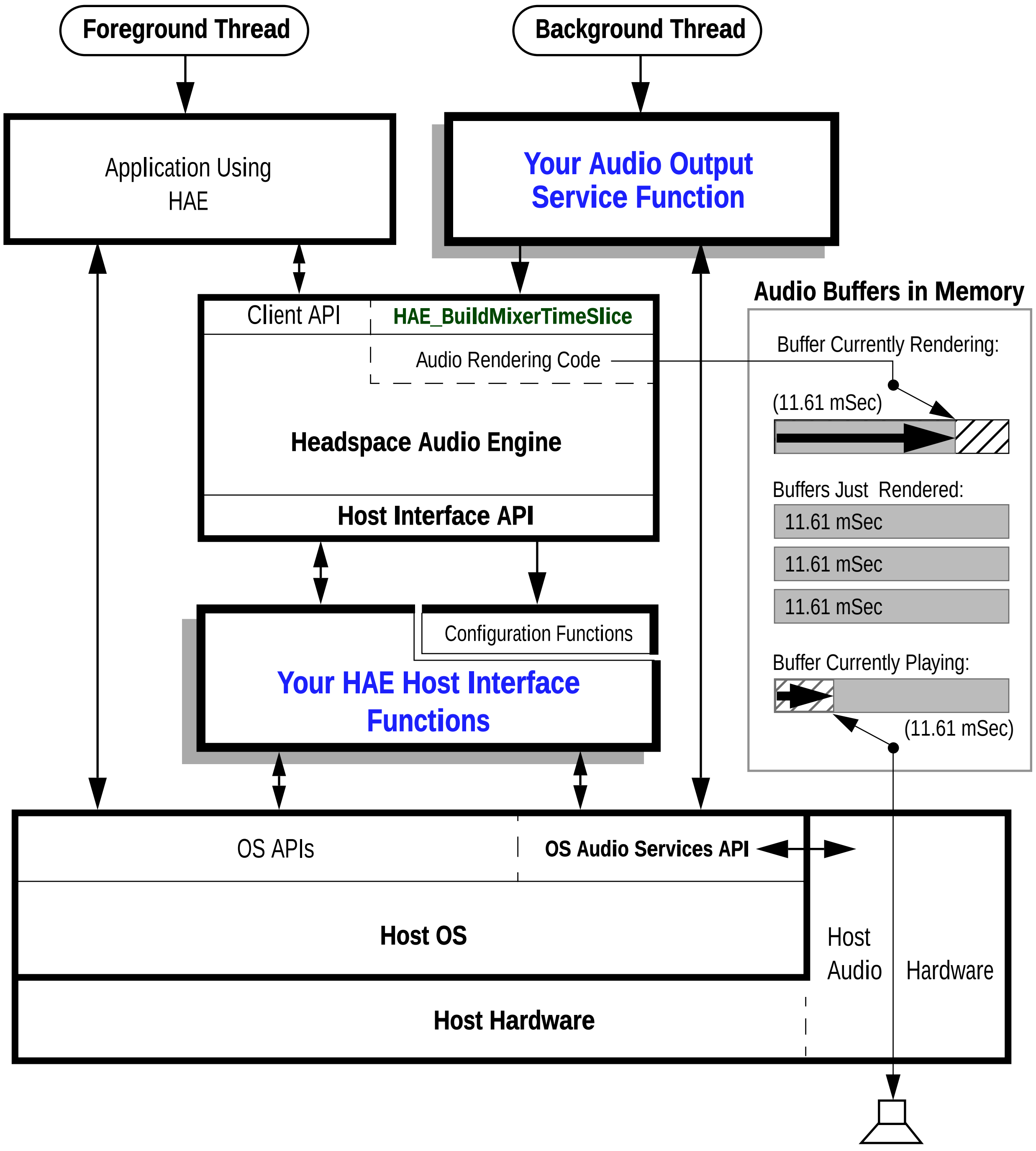
Moving Ahead

Now let’s take a look at where the Host Interface API fits into the runtime scheme, an overview of the structure of the API, and how to use the API.

Programming with the HAE Host Interface API

A Technical Overview of HAE Installations

Here, in a nutshell, is where the functions you write will fit into the HAE scheme at runtime:



As the diagram indicates, there are two forks to an HAE Host Interface installation:

The background process of keeping the digital audio output streams fed with fresh data, and

The interface to various host OS services that HAE needs

Generating Audio Output Streams

The point of the HAE library is to generate an output stream of digital audio data. Although this stream may consist of silence some of the time, at runtime HAE will only function if it is able to continually pump digital audio data samples out to the host– and it can't do that without your help.

In creating a runtime environment for HAE, you must do whatever the host system requires you to do to see that the HAE Host Interface API function **HAE_BuildMixerTimeSlice** is called whenever the host system needs more HAE output– i.e., whenever the sound hardware or host audio software is about to reach the end of the buffer it's currently sending to the audio output jack. Typically, this means setting up a background process thread[1] that calls an audio output service function, which you must also write. Further, once **HAE_BuildMixerTimeSlice** returns to your audio output service function and you have one or more new buffers of HAE output data, your function must also do whatever is necessary in your host system's scheme to see that the new data is connected to the host's audio hardware or audio software output facilities at the right time.

Host OS Services Interface

Everything else in the Host Interface API– that is, all of the other functions that you must provide– can be seen as existing only to support your calls to **HAE_BuildMixerTimeSlice**. HAE calls these host OS functions when in the course of handling user commands[2] or in the course of rendering audio buffers[3] it needs to use services best obtained from the OS: files, memory, time, and services related to audio output devices. (Note that the host OS interface calls may be invoked at any time, including from the background thread.) The API also includes initialization and termination callbacks.

Audio Output Streams in Detail

HAE Output Buffer Sizes and Time Slice Durations

As with most digital audio systems, HAE emits its logically continuous output streams as a discrete series of finite-length buffers of digital audio data, always working ahead of the sound that's currently being heard by the user– which is to say, always rendering now for the future.

When **HAE_BuildMixerTimeSlice** is called from the background thread to render sound output buffers, it honors the sampling rate, number of channels, and sample wordsize that you set via other functions in this API (**HAE_AquireAudioCard**, **HAE_GetSliceTimeInMicroseconds**). HAE comes preconfigured to emit[4] enough audio sample frames per buffer to play for 11.610 milliSeconds[5]. These factors taken together determine the size in bytes of each emitted data buffer[6].

Depending upon your situation, you may find that you want or need to have more than 11.610 milliSeconds created at a time– under Mac OS, for example, 44 milliSeconds would be a far more

1. which is basically invisible to HAE's client applications
2. received from its Client API
3. in response to calls to **HAE_BuildMixerTimeSlice**
4. at each call to **HAE_BuildMixerTimeSlice**
5. at the current sampling rate, whatever that sampling rate happens to be
6. HAE also requires you to do that math and provide it with the buffer size (**HAE_GetAudioByteBufferSize**)

convenient timeslice duration. If you do need more, you can just simply call **HAE_BuildMixerTimeSlice** multiple times in a row, and wait longer in between; just be sure to use the function **HAE_GetAudioBufferCount** to tell HAE what buffer multiple you're using[1]. Note however that because HAE is always rendering into the future, the more buffers you generate per batch, the more latency your HAE implementation will exhibit.

HAE Audio Devices

Some host environments will furnish more than one way to send digital audio to the speakers; for example, in the Headspace Beatnik web browser plug-in for Microsoft Internet Explorer, the HAE environment includes both DirectSound and waveOut. HAE will only generate a single mono or stereo digital audio output stream, but the Host Interface API allows you to switch between multiple audio output devices by supporting the use of Device ID numbers and Device Names.

HAE and Process Threads

As noted above, HAE assumes a dual-thread model, where constant background audio output generation occurs in one thread and more sparse, *ad hoc* calls for specific services arrive in the foreground from client applications. Over time, we have arrived at a simple, useful framework that abstracts host OS process thread services into three calls (which you must write), and we've included those interfaces in this API in case you would like to use them. HAE never calls these functions, it'd only be your HAE installation code that would use them.

An Introduction to the API's Functions

Here's a quick rundown of what a Host Interface API implementation requires, broken down by functional area. The detailed function reference section of this document starts at page 11.

Obtaining HAE Digital Audio Output Data Buffers..page 32

HAE_BuildMixerTimeSlice

Unlike almost every other function in the Host Interface API, this one is defined as part of the HAE library. You don't write this one, you call it. You must call this function every time you want the next buffer of HAE digital audio output data. This is typically done from a separate process thread (or via an interrupt service routine), which you must also set up and manage.

Initialization and Termination Callbacks...page 12

HAE_Setup – HAE_Cleanup

If your HAE environment requires any special setup or teardown steps to bracket the initialization and termination of HAE itself, you can put them in these callbacks; otherwise, just return.

Furnishing HAE with Host System Services

Memory Management Services ....page 13

HAE_Allocate – HAE_Deallocate – HAE_IsBadReadPointer – HAE_SizeOfPointer – HAE_BlockMove

Because HAE allocates and moves memory at runtime, it needs to interface with your memory

1. I.e., continuing with the Mac OS example, every 44 milliSeconds you could call HAE_BuildAudioMixerSlice four times in a row, so long as you set the buffer count to 4 first.

manager. These are the memory manager glue functions, and should be simple.

Timing Services..........page 15

HAE_Microseconds – HAE_WaitMicroseconds

Because HAE is all about the time-based phenomena of music and sound, it needs to know what time it is; and sometimes it needs to delay for a fixed amount of time. These should be simple.

File System Services..........page 16

HAE_CopyFileNameNative – HAE_FileCreate – HAE_FileDelete

HAE_FileOpenForRead – HAE_FileOpenForWrite – HAE_FileOpenForReadWrite – HAE_FileClose

HAE_ReadFile – HAE_WriteFile

HAE_SetFilePosition – HAE_SetFileLength – HAE_GetFilePosition – HAE_GetFileLength

Because HAE may be called upon to read or write disk files, it needs to interface with the host file system. These are the file system glue functions, and should be simple. Note that HAE will use the host system's native path representation, whether string or structure.

Process Thread Services..........page 21

HAE_CreateFrameThread – HAE_DestroyFrameThread – HAE_SleepFrameThread

Headspace engineers have found it convenient in adapting HAE to previous hosts to create a layer of abstraction around the host OS thread system, i.e. these three calls. We've left the interface for these functions in the API just in case you find this idea and mechanism useful– you're not under any obligation to furnish them if you don't need them for your host system, or if you simply don't want to use them. Note that systems without threads (i.e. Mac OS) have no need for this arrangement, and may be able to use interrupts to achieve an effect similar to threads.

Furnishing HAE with Audio Output Services

Audio Output Device Allocation Services..........page 23

HAE_AquireAudioCard – HAE_ReleaseAudioCard

These lower-level functions should obtain and relinquish exclusive access to the indicated audio output device for HAE. Note that HAE abstracts all digital audio output facilities, whether hardware or software, as 'Devices'; see next group.

Audio Output Device Management Services..........page 25

HAE_MaxDevices – HAE_SetDeviceID – HAE_GetDeviceID – HAE_GetDeviceName

In HAE installations that furnish multiple sound output options, you can use Device ID numbers and Device Names to keep them straight, and to allow you to switch between them more cleanly.

Audio Hardware Volume Control Services..........page 28

HAE_GetHardwareVolume – HAE_SetHardwareVolume

HAE furnishes its clients with access to the audio output device's master volume control, if there is one in your system. (Note that for a software audio output device, its 'hardware' volume might actually be implemented in software.) Your functions should allow HAE to set and get this volume control.

Audio Configuration Information Services..page 29

HAE_IsStereoSupported – HAE_Is16BitSupported

HAE_GetSliceTimeInMicroseconds – HAE_GetMaxSamplePerSlice – HAE_GetAudioByteBufferSize

HAE_GetAudioBufferCount – HAE_GetDeviceSamplesPlayedPosition

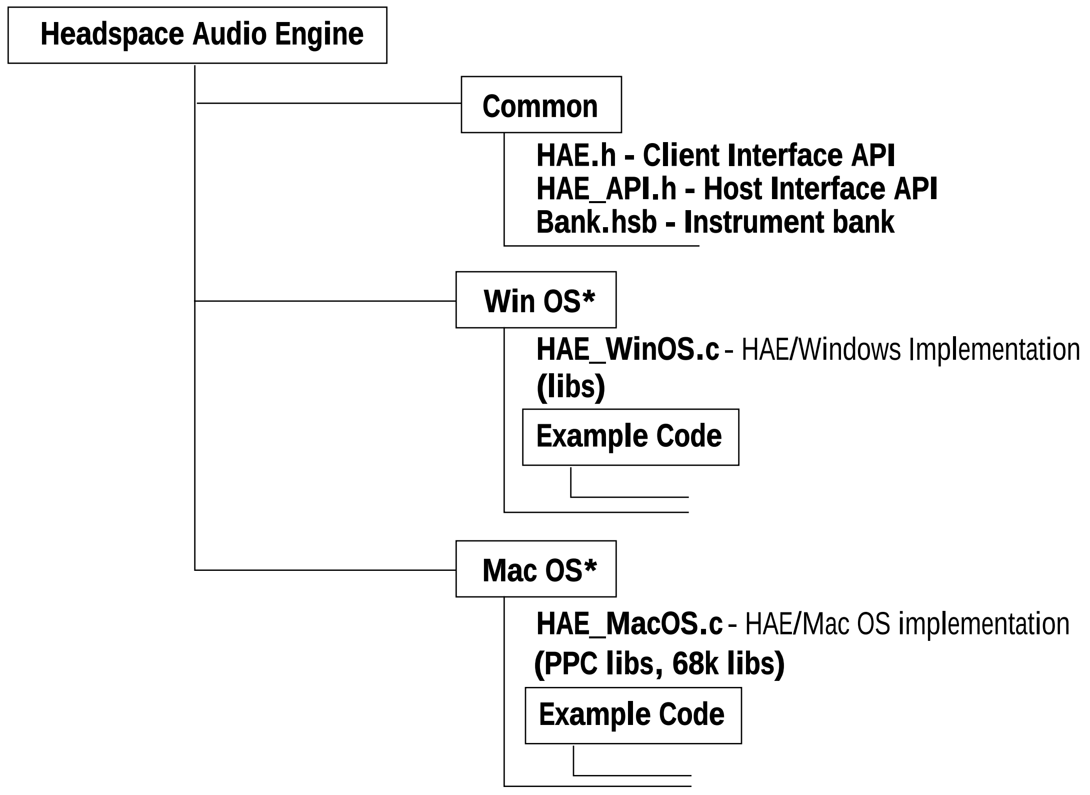
These functions allow HAE to sense its operating environment parameters. They should all be very, very simple one-line functions that return a single value.

Adding the HAE Host Interface API to Your System

For system implementers, HAE is packaged as a library which can be linked with any C or C++ code, and its associated header (interface) file, `HAE_API.h`. HAE been tested with many development systems including Microsoft Visual C version 4 and later, and Metrowerks CodeWarrior version 11 and later. (Note that HAE's simple audio-out requirements mean your development systems don't need to be equipped with fancy, expensive sound cards.)

HAE Release Package Directory Structure

The files are supplied as a Zip or Stuffit archive with the following directory structure (boxes are directories):


Headspace Audio Engine
Common
HAE.h - Client Interface API
HAE_API.h - Host Interface API
Bank.hsb - Instrument bank
Win OS*
HAE_WinOS.c - HAE/Windows Implementation
(libs)
Example Code
Mac OS*
HAE_MacOS.c - HAE/Mac OS implementation
(PPC libs, 68k libs)
Example Code

*These are previous HAE Host Interface implementations that we provide as examples. Other platforms are available upon request, although some may require special nondisclosure arrangements.

Project Settings

HAE and all of its APIs are written to be machine-independent, and is believed to be able to compile in any modern C or C++ compiler. If your HAE Host Interface API functions are correctly built with the same development system and project settings that are used to build the HAE library for your host system, then they should link successfully.

Required Header File: `HAE_API.h`

Note: Your HAE environment will not link successfully unless your HAE support functions conform to the prototypes defined in the header file HAE_API.h. It includes the function prototypes, type definitions, enums, and other declarations and symbol definitions with which the HAE library was built.

Moving Ahead

Next, a detailed look at each function in the API.

HAE Host Interface API Functions Reference

Alphabetical Index to Functions

Subject Index to Functions

Initialization and Termination Callbacks

If your HAE environment requires any special setup or teardown steps to bracket the initialization and termination of HAE itself, you can put them in these callbacks; otherwise, just return.

HAE_Setup

```
int HAE_Setup(void);
```

HAE calls this function while setting up to give you an opportunity to load libraries or DLLs, or to perform any other initialization functions the rest of your HAE support code may require. This occurs before HAE allocates memory or performs any other major initialization steps.

If you don't need to do anything at setup time, just `return(0);`.

Return:

`0` to signal success, or

`-1` to signal an error

See also: HAE_Cleanup, page 12.

HAE_Cleanup

```
int HAE_Cleanup(void);
```

HAE calls this function while closing down to give you an opportunity to unload libraries or DLLs, or to perform any other termination functions the rest of your HAE support code may require. This occurs after HAE deallocates its last audio buffers, and after HAE has released its audio hardware.

If you don't need to do anything at cleanup time, just `return(0);`.

Return:

`0` to signal success, or

`-1` to signal an error

See also: HAE_Setup, page 12.

Memory Management Services

Because HAE allocates and moves memory at runtime, it needs to interface with your memory manager. These are the memory manager glue functions, and should be simple.

HAE_Allocate

```
void* HAE_Allocate(unsigned long size);
```

HAE calls this function when it needs to allocate a locked block of `size` bytes of zero'ed memory. Your function should allocate, lock, and zero the block, and then return a pointer to it.

Return:

A pointer to the freshly allocated block, or

`0` to signal an error

See also: HAE_Deallocate, page 13.

HAE_Deallocate

```
void HAE_Deallocate(void* memoryBlock);
```

HAE calls this function when it no longer needs to use the block of memory starting at `memoryBlock`. The `memoryBlock` pointer parameter is guaranteed to have been obtained from a previous call to **HAE_Allocate**. Your function should free the memory.

Return: (No return value.)

See also: HAE_Allocate, page 13.

HAE_IsBadReadPointer

```
int HAE_IsBadReadPointer(
   void* memoryBlock,
   unsigned long size);
```

HAE calls this function when it needs to know whether it will be able to access a memory block of `size` bytes starting at address `memoryBlock` without causing a memory protection fault.

Return:

`0` to signal that accessing that memory would be OK, or

`1` to signal that the indicated memory cannot be accessed without causing a memory protection fault, or

`2` if you aren’t supporting this test in your HAE installation

See also: ——

HAE_SizeOfPointer

```
unsigned long HAE_SizeOfPointer(void* memoryBlock);
```

HAE calls this function when it needs to know the size in bytes of the block of memory starting at address `memoryBlock`. The `memoryBlock` pointer parameter is guaranteed to have been obtained from a previous call to **HAE_Allocate**.

Return:

The size in bytes of the block of memory, or

`0` if you aren’t supporting this test in your HAE installation

See also: ——

HAE_BlockMove

```
void HAE_BlockMove(
   void* source,
   void* dest,
   unsigned long size);
```

HAE calls this function when it needs to do a block move of memory. In most cases this can be directly implemented by a simple call to the standard C runtime library function `memcpy`; but we have exposed this call in the Host Interface API to allow any systems equipped with hardware block move acceleration (blitters, DMA, etc.) to use it here.

Return: (No return value.)

See also: ——

Timing Services

Because HAE is all about the time-based phenomena of music and sound, it needs to know what time it is; and sometimes it needs to delay for a fixed amount of time. These should be simple.

HAE_Microseconds

```
unsigned long HAE_Microseconds(void);
```

HAE calls this function when it needs to know how long it's been running (or 'what time it is'). Your function should return the number of microseconds elapsed since HAE initialization, to the best reasonable granularity (i.e. time quanta) you are able to provide. HAE prefers a granularity of less than 1000 microSeconds (1 milliSecond), but can deal with a granularity of up to 11,000 microSeconds (11 milliSeconds).

Return:

the number of microseconds elapsed since HAE initialization

See also: HAE_GetDeviceSamplesPlayedPosition, page 31.

HAE_WaitMicroseconds

```
void HAE_WaitMicroseconds(unsigned long wait);
```

HAE calls this function when it needs to wait for the indicated number of microseconds before moving on. Your function should delay that long (i.e. wait or sleep this thread) and then return.

Return: (No return value.)

See also: HAE_SleepFrameThread, page 22.

File System Services

Because HAE may be called upon to read or write disk files, it needs to interface with the host file system. These are the file system glue functions, and should be simple.

A Note Regarding File Path Name Parameters

HAE is designed to work both with host file systems that treat file paths as strings (ala Unix) and with host file systems that treat file paths as structures (ala Mac OS).

For string-based file systems (i.e. Windows, Unix, etc.), all file system path parameters are fully-qualified paths, expressed as null-terminated "C" strings. These paths may be either absolute (starting with a drive spec, i.e. `C:\Dev\MyApp\Sounds\Boom3.WAV`), or relative to the current working directory (i.e. `..\Sounds\Boom3.WAV`).

For structure-based file systems, all file system path parameters are full paths, expressed as pointers to the appropriate native structure (for example, in Mac OS this would be pointers to `FSSpec` structures).

HAE_CopyFileNameNative

```
void HAE_CopyFileNameNative(
   void* pathNameSource,
   void* pathNameDest);
```

HAE calls this function when it needs to make a copy of a file path parameter. `pathNameSource` is guaranteed to be a pointer to a file path parameter previously furnished by one of your functions: either **HAE_FileClose**, **HAE_FileCreate**, **HAE_FileDelete**, **HAE_FileOpenForRead**, **HAE_FileOpenForReadWrite**, or **HAE_FileOpenForWrite.**

Note: HAE makes no promise that there will be enough space available at `pathNameDest` to hold the copy– it is up to your function to ensure that it doesn't copy too far and trash whatever's past the end of the destination block.

Return: (No return value.)

See also: ——

HAE_FileCreate

```
long HAE_FileCreate(void* pathName);
```

HAE calls this function when it needs to create a new disk file at the indicated path. Your function should create the file; if a file with that name already exists, your function should delete it first. It's up to you as an implementer to decide whether to present the user with a warning dialog before deleting the file.

Note: You may wish to consider system security issues when deciding how you wish to implement this function.

Note that opening the file is handled separately; see below.

Return:

`-1` to signal an error, or

`0` to signal success

See also: HAE_FileDelete, page 17.

HAE_FileDelete

```
long HAE_FileDelete(void* pathName);
```

HAE calls this function when it needs to delete the disk file at the indicated path. Your function should delete the file. If the indicated file doesn't exist, that counts as an error and your function should return `-1`. It's up to you as an implementer to decide whether to present the user with a warning dialog before deleting the file.

Note: You may wish to consider system security issues when deciding how you wish to implement this function.

Return:

`-1` to signal an error, or

`0` to signal success

See also: HAE_FileCreate, page 16.

HAE_FileOpenForRead

```
long HAE_FileOpenForRead(void* pathName);
```

HAE calls this function when it needs to open the disk file at the indicated path in preparation for reading its data.

Return:

a file handle to the opened file, or

`-1` to signal an error

See also: HAE_FileClose, page 18.

HAE_FileOpenForWrite

```
long HAE_FileOpenForWrite(void* pathName);
```

HAE calls this function when it needs to open the disk file at the indicated path in preparation for writing data to the file.

Note: You may wish to consider system security issues when deciding how you wish to implement this function.

Return:

a file handle to the opened file, or

`-1` to signal an error

See also: HAE_FileClose, page 18.

HAE_FileOpenForReadWrite

```
long HAE_FileOpenForReadWrite(void* pathName);
```

HAE calls this function when it needs to open the disk file at the indicated path in preparation for both writing data to the file and reading data from the file.

Note: You may wish to consider system security issues when deciding how you wish to implement this function.

Return:

a file handle to the opened file, or

`-1` to signal an error

See also: HAE_FileClose, page 18.

HAE_FileClose

```
void HAE_FileClose(long fileReference);
```

HAE calls this function when it needs to close the indicated, currently open, disk file.

Return: (No return value.)

See also: ——

HAE_ReadFile

```
long HAE_ReadFile(
   long fileReference,
   void* pBuffer,
   long bufferLength);
```

HAE calls this function when it's ready to read the next `bufferLength` bytes from the indicated, currently opened, disk file. Your function should read the data into the block of memory starting at `pBuffer`, and advance the file position by the number of bytes you're able to read.

Return:

The number of bytes actually read from the file (i.e., the last block of data read from the file will almost always be shorter than `bufferLength`), or

`-1` to signal an error

See also: HAE_WriteFile, page 19.

HAE_WriteFile

```
long HAE_WriteFile(
   long fileReference,
   void* pBuffer,
   long bufferLength);
```

HAE calls this function when it's ready to write the next `bufferLength` bytes to the indicated, currently opened, disk file. Your function should write the data from the block of memory starting at `pBuffer`, and advance the file position by the number of bytes you're able to write to the file.

Note: You may wish to consider system security issues when deciding how you wish to implement this function.

Return:

The number of bytes actually written from the file (i.e., the last block of data written to the file will almost always be shorter than `bufferLength`), or

`-1` to signal an error

See also: HAE_ReadFile, page 18.

HAE_SetFilePosition

```
long HAE_SetFilePosition(
   long fileReference,
   unsigned long filePosition);
```

HAE calls this function when it need to set the read/write position of the indicated, currently open, disk file to the indicated number, expressed as a count in bytes from the start of the file.

Return:

`0` to signal success, or

`-1` to signal an error, including an out-of-range parameter

See also: HAE_GetFilePosition, page 19.

HAE_GetFilePosition

```
unsigned long HAE_GetFilePosition(long fileReference);
```

HAE calls this function when it needs to know the current read/write position of the indicated, currently open, disk file.

Return:

The current read/write position of the file, expressed as a count in bytes from the start of the file, or

an error code, in whatever form is native for the host OS file system (for example, under Mac OS the error code for ‘file not open’ is `-38`), or

`0` if your function could not determine the file position but no native error code can be found

See also: HAE_SetFilePosition, page 19.

HAE_GetFileLength

```
unsigned long HAE_GetFileLength(long fileReference);
```

HAE calls this function when it need to know the length of the indicated, currently open, disk file.

Return:

The file’s length in bytes, or

an error code, in whatever form is native for the host OS file system (for example, under Mac OS the error code for ‘file not open’ is `-38`), or

`0` if your function could not determine the file length but no native error code can be found

See also: HAE_SetFileLength, page 20.

HAE_SetFileLength

```
int HAE_SetFileLength(long fileReference, unsigned long newSize);
```

HAE calls this function when it needs to alter the length of the indicated, currently open, disk file.

When increasing the length of a file, Headspace leaves it up to you to decide whether to zero-pad the additional length; HAE does not require you to do so. When decreasing the length of a file, HAE does not require you to truncate the file. In general, whatever your host file system would ordinarily do when changing a file’s length is fine with HAE.

Note: You may however wish to consider system security issues when deciding how you wish to implement this function.

Return:

`0` to signal success, or

`-1` to signal an error

See also: HAE_GetFileLength, page 20.

Process Thread Services

A Note Regarding the Process Thread Functions

Headspace engineers have found it convenient in adapting HAE to previous host systems to create a layer of abstraction around the host OS thread system, i.e. these three calls. We've left the interface for these functions in the API just in case you find this idea and mechanism useful– but unlike the rest of the API **you're not under any obligation to furnish these functions if you don't need them** for your particular situation, or if you simply prefer to handle processes in some other way. Note that systems without threads (i.e. Mac OS) have no need for this arrangement, and may be able to use interrupts to achieve an effect similar to threads.

Also, if you use these functions, you'll need to know about this `typedef`:

```
typedef void* (HAE_FrameThreadProc)(void* context);
```

This defines the type of function to be called by the thread. Note that `context` is a platform-specific value; for example in the JavaSound installation of HAE, it's a `JNIEnv*` pointer.

HAE_CreateFrameThread

```
extern int HAE_CreateFrameThread(
   void* context,
   HAE_FrameThreadProc proc);
```

Your HAE installation should call this when it needs to set up a background process thread to service host system audio output. Your **HAE_CreateFrameThread** function should call whatever host system functions are necessary to create a new process frame thread, attach the indicated procedure to it, and start the thread. Typically `proc` will be the address of your audio output service function. If you're managing multiple threads, provide a different, unique `context` to identify each thread you create; later, when you want to kill or sleep the thread, you can use `context` again to specify which thread you mean.

Return:

`0` to signal success, or

`-1` to signal an error

See also: HAE_DestroyFrameThread, page 21.

HAE_DestroyFrameThread

```
extern int HAE_DestroyFrameThread(void* context);
```

Your HAE installation should call this when it needs to terminate the background process thread that services host system audio output. Your **HAE_DestroyFrameThread** function should call whatever host system functions are necessary to stop and destroy the indicated process frame thread. If you're managing multiple threads, use `context` to identify the thread you want to kill.

Return:

`0` to signal success, or

`-1` to signal an error

See also: HAE_CreateFrameThread, page 21.

HAE_SleepFrameThread

```
extern int HAE_SleepFrameThread(void* context, long msec);
```

Your HAE installation should call this when it needs to make a thread sleep for a given amount of time. For example, your audio output service thread will typically want to go to sleep for a while after generating each new batch of HAE output buffers. Your **HAE_SleepFrameThread** function should call whatever host system functions are necessary to sleep the indicated process frame thread for the indicated number of milliseconds.

Because of the different ways threads are handled under different operating systems, we're leaving our implementation guidelines for this function a little vague. For example, under Windows and Mac OS you can only sleep the thread from which you're calling; if on the other hand you're managing multiple threads under an OS that allows you to selectively sleep threads other than the one you're in, we support a `context` parameter to identify the particular thread you want to sleep.

Return:

`0` to signal success, or

`-1` to signal an error

See also: HAE_WaitMicroseconds, page 15.

Audio Output Device Allocation Services

These lower-level functions should obtain and relinquish exclusive access to the indicated audio output device for HAE. Note that HAE abstracts all digital audio output facilities, whether hardware or software, as ‘Devices’; see next group.

HAE_AquireAudioCard

```
int HAE_AquireAudioCard(
   void* context,
   long sampleRate,
   long channels,
   long bits);
```

HAE calls this function when it needs to begin exclusive access to an audio output device whose capabilities meet or exceed the indicated specifications. Later, when HAE no longer needs to monopolize the device (or indeed to use it at all), HAE will call your **HAE_ReleaseAudioCard** function. Your **HAE_AquireAudioCard** function should save the device’s existing state (if this applicable for your device) so that it can be restored later in **HAE_ReleaseAudioCard**, and also perform any initialization the device may require. You may find the `context` parameter useful for this, or you may prefer to use `context` to keep multiple audio devices straight.

The `sampleRate` parameter is guaranteed to be either `44100`, `22050`, or `11025`;
the `channels` parameter is guaranteed to be either `1` or `2`;
and the `bits` parameter is guaranteed to be either `8` or `16`.

Return:

`0` to signal success, or

`-1` to signal an error

See also: HAE_ReleaseAudioCard, page 23.

HAE_ReleaseAudioCard

```
int HAE_ReleaseAudioCard(void* context);
```

HAE calls this function when it no longer needs to use the audio output device that you allocated to it earlier when HAE called your **HAE_AquireAudioCard** function. If you’ve been allocating multiple audio output devices, you can tell which one HAE is freeing this time by examining the `context` pointer parameter to see if it rings a bell. If your **HAE_AquireAudioCard** function saved the device’s state before HAE acquired it, then your **HAE_ReleaseAudioCard** function should restore that state; you might prefer to use `context` for that instead.

Note that your audio output service thread (or interrupt service routine) might also need to call this function if it decides that something has gone awry and HAE audio output must end.

Return:

`0` to signal success, or

`-1` to signal an error

See also: HAE_AquireAudioCard, page 23.

Audio Output Device Management Services

A Note Regarding Device ID Parameters

If your application calls for sending separate HAE digital audio streams to separate digital audio output devices, you can use the scheme of **`deviceID`**s implemented in the following function interfaces to manage and maintain them. With these functions, you assign each digital audio output device its own unique **`deviceID`**, declare to HAE the number of available digital audio output devices, furnish HAE with a textual description of each device, and handle HAE's requests for the name and **`deviceID`** of the currently selected output device.

Note that, like the **`refCon`** field of a window, **`deviceID`**s are provided for your reference only–HAE itself never uses these values, it just send them back to you.

HAE_MaxDevices

```
long HAE_MaxDevices(void);
```

HAE calls this function when it needs to know the number of separate digital audio output devices you want it to service. For example, if you're servicing both DirectSound and waveOut, you'd return **`2`**.

Note: This function may be called before any other Host Interface API calls.

Return:

the number of devices you're servicing, or

`-1` to signal an error

See also: ——

HAE_SetDeviceID

```
void HAE_SetDeviceID(
   long deviceID,
   void* deviceParameter);
```

HAE calls this function when it needs to set its digital audio output device. Your function should call **HAE_ReleaseAudioCard** for the current device (if there is one yet), and then call **HAE_AquireAudioCard** for the indicated device, in addition to whatever other operations your implementation requires.

`deviceID` is guaranteed to be between **`0`** and **`HAE_MaxDevices()`**. Your function can set **`deviceParameter`** to point at your block of device-specific data for the indicated **`deviceID`**, if applicable.

Note: This function may be called before any other Host Interface API calls.

On Return:

point `deviceParameter` at your block of device-specific data for the indicated `deviceID`, if you're using any, or

set `deviceParameter` to `NULL`

See also: HAE_GetDeviceID, page 26.

HAE_GetDeviceID

```
long HAE_GetDeviceID(void* deviceParameter);
```

HAE calls this function when it needs to obtain the `deviceID` of the digital audio output device you want it to service. In addition to returning the `deviceID`, your function should also either point `deviceParameter` at your block of device-specific data for the current `deviceID`, or set `deviceParameter` to `NULL` if you aren't using the `deviceParameter` facilities.

Note: This function may be called before any other Host Interface API calls.

Return:

the `deviceID` of the device currently selected, or

`-1` to signal an error

And:

point `deviceParameter` at your block of device-specific data for the current `deviceID`, if you're using any, or

set `deviceParameter` to `NULL`

See also: HAE_SetDeviceID, page 25.

HAE_GetDeviceName

```
void HAE_GetDeviceName(
    long deviceID,
    char* cName,
    unsigned long cNameLength);
```

HAE calls this function when it needs to obtain the name string you wish to associate with the indicated `deviceID`. Your function should furnish a unique and appropriate i.d. string for the indicated device, not more that `cNameLength` bytes long.

For future compatibility, this string you provide should be of the form `"platform,method,misc"` where `platform` describes the host OS or machine, and `method` and `misc` together describe the audio output system and runtime environment used.

For example, previous HAE implementations have used the following device names:

```
"MacOS,Sound Manager 3.0,SndPlayDoubleBuffer"
"WinOS,DirectSound,multi threaded"
"WinOS,waveOut,multi threaded"
"WinOS,VxD,low level hardware"
```

    "WinOS,plugin,Director"

Note, however, that currently nobody examines the contents of these strings; all that matters to HAE is that each device name string you furnish is unique.

Note: This function may be called before any other Host Interface API calls.

Return:

The identifying string for the indicated `deviceID`, expressed as a pointer to a null-terminated C string

See also: —–

Audio Hardware Volume Control Services

HAE_GetHardwareVolume

```
short int HAE_GetHardwareVolume(void);
```

HAE calls this function when it needs to determine the current setting of the volume control for the audio output environment. Note that depending upon your host system's design, the volume control may be implemented in software, in hardware, or both.

Return:

A fixed-point fractional number ranging from `0` (`0x0000`) to `256` (`0x0100`), with `0` meaning minimum possible volume control setting (i.e. complete silence), and `256` corresponding to maximum possible volume control setting. If your machine has no output volume control, always return `256`. If your machine uses a volume control range other than `0..256`, scale that range to `0..256` and return the result.

See also: HAE_SetHardwareVolume, page 28.

HAE_SetHardwareVolume

```
void HAE_SetHardwareVolume(short int theVolume);
```

HAE calls this function when it needs to set the volume control for the audio output environment. `theVolume` is a fixed-point fractional number ranging from `0` (`0x0000`) to `256` (`0x0100`), with `0` meaning minimum possible volume control setting (i.e. complete silence), and `256` corresponding to maximum possible volume control setting. If your machine uses a volume control range other than `0..256`, scale `theVolume` to that range. Note that depending upon your host system's design, the volume control may be implemented in software, in hardware, or both.

If your machine has no output volume control, your function can simply return without doing any work.

Return: (No return value.)

See also: HAE_GetHardwareVolume, page 28.

Audio Configuration Information Services

These functions allow HAE to sense its operating environment parameters. They should all be very, very simple one-line functions that return a single value.

HAE_IsStereoSupported

```
int HAE_IsStereoSupported(void);
```

HAE calls this function when it needs to determine whether the audio output environment in which it's running is mono or stereo. Your function should return `1` for stereo or `0` for mono. Stereo host system software driving mono audio hardware would mean mono; Mono host system software driving stereo audio hardware would also mean mono.

Return:

`1` if the host audio system permits stereo, or

`0` for mono

See also: ——

HAE_Is16BitSupported

```
int HAE_Is16BitSupported(void);
```

HAE calls this function when it needs to determine whether the audio output environment in which it's running is 8-bit or 16-bit. Your function should return `1` for 16-bit or `0` for 8-bit. 16-bit host system software driving 8-bit audio hardware would mean 8-bit; 8-bit host system software driving 16-bit audio hardware would also mean 8-bit.)

Return:

`1` if the host audio system supports 16 bit audio, or

`0` for 8 bit audio

See also: ——

HAE_GetSliceTimeInMicroseconds

```
unsigned long HAE_GetSliceTimeInMicroseconds(void);
```

HAE calls this function when it needs to know the amount of real time of audio output data that you want HAE to generate in each audio buffer, i.e. each time your audio output service function calls **HAE_BuildMixerTimeSlice**. Note that you can (and typically will) use **HAE_GetAudioBufferCount** to set **HAE_BuildMixerTimeSlice** to generate multiple buffers of this duration at a batch, not just one buffer at each call.

Return:

Your desired duration in microseconds; this is typically `11,610`.

See also: ——

HAE_GetAudioBufferCount

```
int HAE_GetAudioBufferCount(void);
```

HAE calls this when it needs to know the number of audio output buffer blocks you want it to generate each time you call **HAE_BuildMixerTimeSlice**.

This number should be chosen carefully, as it affects both the distribution over time of HAE's CPU load and the overall audio latency of your HAE installation. For example, if you were to set **HAE_BuildMixerTimeSlice** to create 8 buffers at each call, then you'd get larger but more widely-spaced peaks of CPU usage[1], but the latency will increase because HAE would always be rendering the audio anywhere from 1 to 8 buffers into the future[2].

Note that you can calculate your HAE installation's audio latency by multiplying the **HAE_GetAudioBufferCount** figure by the **HAE_GetSliceTimeInMicroseconds** figure.

Note: You still have to call **HAE_BuildMixerTimeSlice** the indicated number of times yourself in each batch– that call always generates exactly one buffer per call.

Return:

The number of buffer blocks you'll be generating in each batch.

See also: ——

HAE_GetAudioByteBufferSize

```
long HAE_GetAudioByteBufferSize(void);
```

HAE calls this function when it needs to know how big each final audio output buffer is.

You can determine this number of bytes by the following formula:

size = sliceTimeInSeconds * samplingRate * numberOfChannels * bytesPerSample

Note that if none of these factors will be changing at runtime in your HAE environment, you can calculate the size number at compile time rather than doing math in the target machine.

Note: Several calls in the Host Interface API deal with the buffer size and the buffer sample frame count; your functions **must** consistently use the same values everywhere (**HAE_GetAudioByteBufferSize**, **HAE_GetMaxSamplePerSlice**, and **HAE_BuildMixerTimeSlice**).

Return:

1. which can be either good or bad depending on how it affects your other CPU usages
2. which is almost always bad, especially in applications like games or web browsers where the audio content is interactive

The audio output buffer’s size in bytes.

See also: ——

HAE_GetMaxSamplePerSlice

```
extern short HAE_GetMaxSamplePerSlice(void);
```

HAE calls this function when it needs to know the number of sample frames (not bytes, not words, not samples) it needs to generate to fill each audio output buffer. (In HAE, there is one audio output buffer per time slice.)

You can determine this number of sample frames by the following formula:

sampleFramesPerSlice = sliceTimeInSeconds * samplingRate

Note that if neither of these factors will be changing at runtime in your HAE environment, you can calculate the sample frames number at compile time rather than doing math in the target machine.

Note: Several calls in the Host Interface API deal with the buffer size and the buffer sample frame count; your functions **must** consistently use the same values everywhere (**HAE_GetAudioByteBufferSize**, **HAE_GetMaxSamplePerSlice**, and **HAE_BuildMixerTimeSlice**).

Return:

The number of sample frames it takes (at the current sample rate) to play sound for the duration of a time slice.

See also: ——

HAE_GetDeviceSamplesPlayedPosition

```
unsigned long HAE_GetDeviceSamplesPlayedPosition(void);
```

HAE calls this function when it needs to know how many sample frames (not samples, not bytes, not buffers) have been emitted to the digital audio output device since HAE initialization. (Note that even when no audio program material is being generated, HAE continually pumps out sample frames filled with silence.)

Return:

The number of sample frames emitted to the digital audio output device since HAE initialization.

See also: HAE_Microseconds, page 15.

Obtaining HAE Digital Audio Output Data

HAE_BuildMixerTimeSlice

```
extern void HAE_BuildMixerSlice(
   void* context,
   void* pAudioBuffer,
   long bufferByteLength,
   long sampleFrames);
```

Note: Unlike almost every other function described in this document, this function is furnished for you in the HAE library. You don't have to write it, but you do need to call it.

Your HAE installation must call this function every time the host audio output system[1] is ready for a new buffer of HAE audio output data.

You must furnish the address of an output buffer that you want HAE to use (`pAudioBuffer`) and the number of sample frames[2] (not bytes, not words, not samples) that you want HAE to put there (`sampleFrames`); HAE will write to that buffer until it's filled up your indicated number of bytes (`bufferByteLength`)[3]. Your `sampleFrames` and `bufferByteLength` parameters should agree.

Note: Several calls in the Host Interface API deal with the buffer size and the buffer sample frame count; your functions **must** consistently use the same values everywhere (**HAE_GetAudioByteBufferSize**, **HAE_GetMaxSamplePerSlice**, and **HAE_BuildMixerTimeSlice**).

You can determine the amount of audio realtime generated in each call to **HAE_BuildMixerSlice** by the following formula:

$$\frac{(((\text{buffer size in bytes}) / (\text{bytes per sample})) / (\text{number of channels}))}{(\text{sampling rate in Hz})}$$

Return: (No return value.)

See also: ——

[End of Functions Reference]

1. which may be a sound card, other audio hardware, or system software such as a sound driver
2. For mono, one sample frame is a single sample; for stereo, a sample frame is a single left sample and a single right sample; note that these may be either 8-bit samples or 16-bit samples, as previously set with **HAE_AquireAudioCard**.
3. At present, the `bufferByteLength` parameter is ignored by this call; however, it should be set to either `256`, `512`, `1024`, or `2048`, and should agree with what you specified previously via **HAE_GetAudioByteBufferSize**.

Index