OpenAL 1.1 Specification and Reference

Specification and Reference

This is the OpenAL 1.1 Specification and Reference. This document is based upon the older OpenAL Specification and Reference (1.0), published in June 2000. Both copyright notices are presented below.

Version 1.1
Published June 2005

Version 1.0 Draft Edition
Published June 2000

Table of Contents

1. Introduction

1.1. Revision History, 1.1 Document

Revision 1.0 January 2005 Garin Hiebert
First draft of new specification, submitted to 1.1 list for review.
Revision 1.1 February 2005 Garin Hiebert
Incorporated feedback from Ryan Gordon.
Revision 1.2 February 2005 Garin Hiebert
Incorporated feedback from Ryan Gordon, Daniel Peacock, and Carlo Vogelsang.
Revision 1.3 February 2005 Garin Hiebert
New set of revisions before public release of draft.
Revision 1.4 February 2005 Garin Hiebert
Incorporated feedback from Andrew McDonald, Ryan Gordon, Joe Tennies, and Joseph Valenzuela.
Revision 1.5 March 2005 Garin Hiebert
Incorporated feedback from many openal-devel list members.
Revision 1.6 March 2005 Garin Hiebert
Incorporated feedback from many openal-devel list members.
Revision 1.7 April 2005 Garin Hiebert
Incorporated feedback from many openal-devel list members.
Revision 1.8 April 2005 Garin Hiebert
Incorporated feedback from many openal-devel list members.
Revision 1.9 April 2005 Garin Hiebert
Incorporated feedback from Jean-Marc Jot, Daniel Peacock, and Jean-Michel Trivi.
Revision 2.0 May 2005 Garin Hiebert
Incorporated feedback from many openal-devel list members.
Revision 2.1 May 2005 Garin Hiebert
Incorporated feedback from Bob Aron, Alexandre Mah, and Carlo Vogelsang.
Revision 2.2 May 2005 Garin Hiebert
Added AL_SOURCE_TYPE attribute and description.
Revision 2.3 June 2005 Garin Hiebert
Added feedback from Alexandre Mah and Daniel Peacock.
Revision 2.4 June 2005 Garin Hiebert
Removed “Draft” designation.
Revision 2.5 July 2005 Garin Hiebert
Incorporated feedback from Sven Panne, and made minor corrections to the content and formatting.
Revision 2.6 August 2005 Garin Hiebert
Incorporated feedback from Stephen Baker and Sven Panne.
Revision 2.7 October 2005 Garin Hiebert
Incorporated feedback from Bob Aron, Sven Panne, and Eric Wing.
Revision 2.8 June 2006 Garin Hiebert
Incorporated feedback from Bob Aron, Nathan Charles, Sven Panne, and made minor corrections to the content and formatting.
Revision 2.9 July 2006 Garin Hiebert
Incorporated feedback from Sven Panne regarding capture support.

1.2. A Brief History of OpenAL

The first discussions about implementing OpenAL as an audio API complementary to OpenGL started around 1998. There were a few aborted attempts at creating the headers and a specification, but by late 1999 Loki Entertainment Software was in need for an API of exactly this type and pursued both a specification and a Linux implementation. At around that time, Loki started talking with Creative Labs about standardizing the API and expanding platform support. The OpenAL 1.0 specification was released in early 2000 and compliant OpenAL libraries were released in the same year for Linux, MacOS 8/9, Windows, and BeOS. Loki Entertainment also shipped several games using OpenAL in 2000 – Heavy Gear 2 and Heretic 2 (both under Linux). In 2001, Creative Labs released the first hardware-accelerated OpenAL libraries. The libraries supported the SoundBlaster Live on MacOS 8/9 and Windows.

Since 2001, there has been continuous improvement in OpenAL. Some platforms are less relevant than in 2000 (BeOS and MacOS 8/9 for instance), but more platforms have been added as well (BSD, Solaris, IRIX, Mac OS X, and the popular console gaming platforms). Hardware support is enabled for many Creative and NVIDIA audio devices under Windows as well.

In terms of product support, OpenAL has been used in a large number of titles over the years, on many platforms (for a list of many of the titles, see http://www.openal.org/titles.html).

1.3. What Is the OpenAL Audio System?

OpenAL (for "Open Audio Library") is a software interface to audio hardware. The interface consists of a number of functions that allow a programmer to specify the objects and operations in producing high-quality audio output, specifically multichannel output of 3D arrangements of sound sources around a listener.

The OpenAL API is designed to be cross-platform and easy to use. It resembles the OpenGL API in coding style and conventions. OpenAL uses a syntax resembling that of OpenGL where applicable.

OpenAL is foremost a means to generate audio in a simulated three-dimensional space. Consequently, legacy audio concepts such as panning and left/right channels are not directly supported. OpenAL does include extensions compatible with the IA-SIG 3D Level 1 and Level 2 rendering guidelines to handle sound-source directivity and distance-related attenuation and Doppler effects, as well as environmental effects such as reflection, obstruction, transmission, and reverberation.

Like OpenGL, the OpenAL core API has no notion of an explicit rendering context, and operates on an implied current OpenAL Context. Unlike the OpenGL specification the OpenAL specification includes both the core API (the actual OpenAL API) and the

operating system bindings of the ALC API (the "Audio Library Context"). Unlike OpenGL's GLX, WGL and other OS-specific bindings, the ALC API is portable across platforms as well.

1.4. Differences Between OpenAL 1.1 and OpenAL 1.0

1.4.1. Recording API

OpenAL 1.1 implementations support recording as specified in sections 6.4.2 and 9.1.

1.4.2. Get/Set Offset

OpenAL 1.1 implementations support offset operations as specified in sections 4.3.2 and 9.2.

1.4.3. Linear Distance Models

OpenAL 1.1 implementations will support two new linear distance models as specified in sections 3.4.3 and 9.3.

1.4.4. Exponential Distance Models

OpenAL 1.1 implementations will support two new exponential distance models as specified in sections 3.4.5 and 9.4.

1.4.5. Doppler

Doppler behavior has been standardized for OpenAL 1.1, as documented in section 3.5.2.

1.4.6. Mono/Stereo Hints

Hints can now be provided at context-creation time to indicate the number of mono or stereo sources desired for that context, as documented in section 6.2.1.

1.4.7. Standard Extensions Listings

Extension listing behavior has been standardized for OpenAL 1.1, as documented in section 6.3.3. Also, extension names passed to alIsExtensionPresent or alcIsExtensionPresent are no longer case sensitive. Internally to the implementation, the names will be maintained as all upper-case, and when names are expressed by the implementation they will be expressed as all upper-case.

1.4.8. Standard Suspend/Process Behavior

Context suspend/process behavior has been clarified in section 6.2.5.

1.4.9. ALUT Revisions

The old ALUT functions will continue to be supported on platforms where they were traditionally included within the OpenAL library (non-Windows platforms), but a new standalone library will be developed under a separate specification. Everyone will be encouraged to use the new library.

1.4.10. Streaming Clarifications

Streaming using OpenAL's queuing mechanism has been clarified in section 4.3.5.

1.4.11. Error Codes

Error codes are specified in many cases where the old specification was vague.

1.4.12. Pitch Shifting Limits

The pitch shifting limits for OpenAL 1.1 have changed as specified in section 4.3.2.

1.4.13. New ALchar and ALCchar types

New ALchar and ALCchar types have been added, affecting the following functions: alGetString, alIsExtensionPresent, alGetProcAddress, alGetEnumValue, alcOpenDevice, alcIsExtensionPresent, alcGetProcAddress, alcGetEnumValue, and alcGetString.

1.4.14. alcCloseDevice Return Value

alcCloseDevice now returns ALCboolean to indicate success or failure.

1.4.15. Versioning Changes

Clearer definitions of the AL_VERSION, AL_RENDERER, and AL_VENDOR string queries are defined.

1.5. Programmer's View of OpenAL

To the programmer, OpenAL is a set of commands that allow the specification of sound sources and a listener in three dimensions, combined with commands that control how these sound sources are rendered into the output buffer. The effect of OpenAL commands is not guaranteed to be immediate, as there are latencies depending on the implementation, but ideally such latency should not be noticeable to the user.

A typical program that uses OpenAL begins with calls to open a sound device which is used to process output and play it on attached hardware (speakers or headphones). Then, calls are made to allocate an AL context and associate it with the device. Once an AL context is allocated, the programmer is free to issue AL commands. Some calls are used to render sources (point and directional sources, looping or not), while others affect the rendering of these sources including how they are attenuated by distance and relative orientation.

1.6. Implementor's View of OpenAL

To the implementor, OpenAL is a set of commands that affect the operation of CPU and sound hardware. If the hardware consists only of an addressable output buffer, then OpenAL must be implemented almost entirely on the host CPU. In some cases audio hardware provides DSP-based and other acceleration in various degrees. The OpenAL implementor's task is to provide the CPU software interface while dividing the work for each AL command between the CPU and the audio hardware. This division should be tailored to the available audio hardware to obtain optimum performance in carrying out AL calls.

OpenAL maintains a considerable amount of state information. This state controls how the sources are rendered into the output buffer. Some of this state is directly available to the user: he or she can make calls to obtain its value. Some of it, however, is visible only by the effect it has on what is rendered. One of the main goals of this specification is to make OpenAL state information explicit, to elucidate how it changes, and to indicate what its effects are.

1.7. Our View

We view OpenAL as a state machine that controls a multichannel processing system to synthesize a digital stream, passing sample data through a chain of parametrized digital audio signal processing operations. This model should engender a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. Any proper implementation must produce results conforming to those produced by the specified methods, but there may be ways to carry out a particular computation that are more efficient than the one specified.

1.8. Requirements, Conformance and Extensions

The specification has to guarantee a minimum number of resources. However, implementations are encouraged to compete on performance, available resources, and output quality.

There is a set of conformance tests available along with the open source sample implementation. Vendors and individuals are encouraged to specify and implement extensions to OpenAL. Successful extensions will become part of the core specification as necessary and desirable. OpenAL implementations have to guarantee backwards compatibility and ABI compatibility for minor revisions.

The current sample implementation and documentation for OpenAL can be obtained from http://www.openal.org/.

1.9. Architecture Review and Acknowledgments

Like OpenGL, OpenAL is meant to evolve through a joined effort of implementors and application programmers meeting in regular sessions of an Architecture Review Board (ARB). As of this time an ARB has not yet been set up.

Consequently OpenAL has been a long-term informal cooperative effort. The following list (in all likelihood incomplete) gives in alphabetical order participants in the discussion and contributors to the specification processes and related efforts:

Bob Aron, Juan Carlos Arevalo Baeza, Jonathan Blow, Nathan Charles, Keith Charley, Scott Draeker, Ryan Gordon, John Grantham, Jacob Hawley, Garin Hiebert, Carlos Hasan, Nathan Hill, Stephen Holmes, Bill Huey, Mike Jarosh, Jean-Marc Jot, Maxim Kizub, John Kraft, Bernd Kreimeier, Eric Lengyel, Alexandre Mah, Andrew McDonald, Adam Moss, Ian Ollmann, Rick Overman, Sean L. Palmer, Sven Panne, Daniel Peacock, Pierre Phaneuf, Terry Sikes, Joe Tennies, Jean-Michel Trivi, Joseph Valenzuela, Michael Vance, Carlo Vogelsang

2. OpenAL Operation

2.1. OpenAL Fundamentals

OpenAL is concerned with rendering audio into an output buffer and collecting audio data from input buffers. OpenAL's primary use is assumed to be for spatialized sample-based audio. There is no support for MIDI.

OpenAL has three fundamental primitives or objects: buffers, sources, and a single listener. Each object can be changed independently; the setting of one object does not affect the setting of others. The application can also set modes that affect processing. Modes are set, objects specified, and other OpenAL operations performed by sending commands in the form of function or procedure calls.

Sources store locations, directions, and other attributes of an object in 3D space and have a buffer associated with them for playback. When the program wants to play a sound, it controls execution through a source object. Sources are processed independently from each other.

Buffers store compressed or uncompressed audio data. It is common to initialize a large set of buffers when the program first starts (or at non-critical times during execution -- between levels in a game, for instance). Buffers are referred to by sources. Data (audio sample data) is associated with buffers.

There is only one listener (per audio context). The listener attributes are similar to source attributes, but are used to represent where the user is hearing the audio from. All the sources are rendered (mixed and played) relative to the listener.

2.2. Primitive Types

The OpenAL library primitive (scalar) data types mimic the OpenGL data types, allowing seamless integration with OpenGL code. Guaranteed minimum sizes are stated for OpenGL data types, but the actual choice of C data type is left to the implementation. All implementations on a given binary architecture, however, must use a common definition of these data types.

Table 2.1: AL Primitive Data Types

AL Type	*Description*	*GL Type*
ALboolean	8-bit boolean	GLboolean
ALchar	character	GLchar
ALbyte	signed 8-bit 2's-complement integer	GLbyte
ALubyte	unsigned 8-bit integer	GLubyte
ALshort	signed 16-bit 2's-complement integer	GLshort
ALushort	unsigned 16-bit integer	GLushort
ALint	signed 32-bit 2's-complement integer	GLint
ALuint	unsigned 32-bit integer	GLuint
ALsizei	non-negative 32-bit binary integer size	GLsizei
ALenum	enumerated 32-bit value	GLenum
ALfloat	32-bit IEEE 754 floating-point	GLfloat
ALdouble	64-bit IEEE 754 floating-point	GLdouble

2.3. Floating-Point Computation

Any representable floating-point value is legal as input to an OpenAL command that requires floating point data. The result of providing a value that is not a floating point number to such a command is unspecified, but must not lead to OpenAL being interrupted or terminated. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to an OpenAL command yields predictable results, while providing a NaN or infinity yields unspecified results.

Some calculations require division. In such cases (including implied divisions required by vector normalizations), a division by zero produces an unspecified result but must not lead to OpenAL interruption or termination.

2.4. AL State

OpenAL maintains considerable state. This document enumerates each state variable and describes how each variable can be changed. For purposes of discussion, state variables are categorized somewhat arbitrarily by their function. For example, although we describe operations that OpenAL performs on the implied output buffer, the output buffer is not part of the OpenAL's state. Certain states of OpenAL objects (e.g. buffer states with respect to queuing) are introduced for discussion purposes, but not exposed through the API.

2.5. AL Command Syntax

OpenAL's commands are functions or procedures. Various groups of commands perform the same operation but differ in how arguments are supplied to them. To conveniently accommodate this variation, we adopt the OpenGL notation for describing commands and their arguments.

2.6. Basic AL Operation

OpenAL can be used for a variety of audio playback tasks, and is an excellent complement to OpenGL for real-time rendering. A programmer who is familiar with OpenGL will immediately notice the similarities between the two APIs in that they describe their 3D environments using similar methods.

For an OpenGL/OpenAL program, most of the audio programming will be in two places in the code: initialization of the program, and the rendering loop. An OpenGL/OpenAL program will typically contain a section where the graphics and audio systems are initialized, although it may be spread into multiple functions. For OpenAL, initialization normally consists of creating a context, creating the initial set of buffers, loading the buffers with sample data, creating sources, attaching buffers to sources, setting locations and directions for the listener and sources, and setting the initial values for state global to OpenAL.

Initialization Example:

```
// Initialize Open AL
device = alcOpenDevice(NULL); // open default device
if (device != NULL) {
        context=alcCreateContext(device,NULL); // create context
        if (context != NULL) {
                alcMakeContextCurrent(context); // set active context
        }
}
```

The audio update within the rendering loop normally consists of telling OpenAL the current locations of the sources and listener, updating the environment settings, and managing buffers.

Processing Loop Example:

```
// PlaceCamera - places OpenGL camera & updates OpenAL listener data
void AVEnvironment::PlaceCamera()
{
   // update OpenGL camera position
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glFrustum(-0.1333, 0.1333, -0.1, 0.1, 0.2, 50.0);

   gluLookAt(listenerPos[0], listenerPos[1], listenerPos[2],
        (listenerPos[0] + sin(listenerAngle)), listenerPos[1],
        (listenerPos[2] - cos(listenerAngle)),
        0.0, 1.0, 0.0);

   // update OpenAL
```

```
    // place listener at camera
    alListener3f(AL_POSITION, listenerPos[0], listenerPos[1], listenerPos[2]);

    float directionvect[6];
    directionvect[0] = (float) sin(listenerAngle);
    directionvect[1] = 0;
    directionvect[2] = (float) cos(listenerAngle);
    directionvect[3] = 0;
    directionvect[4] = 1;
    directionvect[5] = 0;
    alListenerfv(AL_ORIENTATION, directionvect);
}
```

2.7. AL Errors

OpenAL detects only a subset of those conditions that could be considered errors. This is because in many cases error checking would adversely impact the performance of an error-free program. The command:

ALenum alGetError(void)

is used to obtain error information. Each detectable error is assigned a numeric code. When an error is detected by AL, a flag is set and the error code is recorded. Further errors, if they occur, do not affect this recorded code. When alGetError is called, the code is returned and the flag is cleared, so that a further error will again record its code. If a call to alGetError returns AL_NO_ERROR then there has been no detectable error since the last call to alGetError (or since OpenAL was initialized).

Error codes can be mapped to strings. The alGetString function returns a pointer to a constant (literal) string that is identical to the identifier used for the enumeration value, as defined in the specification.

Table 2.2: Error Conditions

Name	***Description***
AL_NO_ERROR	There is no current error.
AL_INVALID_NAME	Invalid name parameter.
AL_INVALID_ENUM	Invalid parameter.
AL_INVALID_VALUE	Invalid enum parameter value.
AL_INVALID_OPERATION	Illegal call.
AL_OUT_OF_MEMORY	Unable to allocate memory.

The table above summarizes the OpenAL error conditions. When an error flag is set, results of AL operations are undefined only if AL_OUT_OF_MEMORY has occurred. In other cases, the command generating the error is ignored so that it has no effect on AL state or output buffer contents. If the error generating command returns a value, it returns zero. If the generating command modifies values through a pointer argument, no change is made to these values. These error semantics apply only to AL errors, not to system

errors such as memory access errors.

Several error generation conditions are implicit in the description of the various AL commands. First, if a command that requires an enumerated value is passed a value that is not one of those specified as allowable for that command, the error AL_INVALID_ENUM results. This is the case even if the argument is a pointer to a symbolic constant if that value is not allowable for the given command. This will occur whether the value is allowable for other functions, or an invalid integer value.

Integer parameters that are used as names for OpenAL objects such as buffers and sources are checked for validity. If an invalid name parameter is specified in an OpenAL command, an AL_INVALID_NAME error will be generated and the command is ignored.

An attempt to set integral or floating point values out of the specified range will result in the error AL_INVALID_VALUE. The specification does not guarantee that the implementation emits AL_INVALID_VALUE if a NaN or infinity value is passed in for a float or double argument (as the specification does not enforce possibly expensive testing of floating point values).

Commands can be invalid. For example, certain commands might not be applicable to a given object. There are also illegal combinations of tokens and values as arguments to a command. OpenAL responds to any such illegal command with an AL_INVALID_OPERATION error.

If memory is exhausted as a side effect of the execution of an AL command, either on system level or by exhausting the allocated resources at AL's internal disposal, the error AL_OUT_OF_MEMORY may be generated. This can also happen independent of recent commands if OpenAL has to request memory for an internal task and fails to allocate the required memory from the operating system.

Otherwise errors are generated only for conditions that are explicitly described in this specification.

2.8. Controlling AL Execution

The application can temporarily disable certain AL capabilities on a per-Context basis. This allows the driver implementation to optimize for certain subsets of operations. Enabling and disabling capabilities is handled using a function pair.

void alEnable (ALenum target);
void alDisable (ALenum target);

The application can also query whether a given capability is currently enabled or not.

ALboolean alIsEnabled (ALenum target);

If the token used to specify target is not legal, an AL_INVALID_ENUM error will be generated.

2.9. Object Paradigm

OpenAL is an object-oriented API, but it does not expose classes, structs, or other explicit data structures to the application.

2.9.1. Object Categories

OpenAL has three primary categories of objects:

- one unique listener per context
- multiple buffers shared among all contexts (for one device)
- multiple sources, each local to a context

2.10. Static vs. Dynamic Objects

The vast majority of OpenAL objects are dynamic, and will be created on application demand. There are also OpenAL objects that do not have to be created, and can not be created, on application demand. Currently, the listener is the only such static object in OpenAL.

2.11. Object Names

Dynamic objects are manipulated using an integer, which in analogy to OpenGL is referred to as the object's "name". These are of type unsigned integer (ALuint). Names can be valid beyond the lifetime of the context they were requested if the objects in question can be shared among contexts. No guarantees or assumptions are made in the specification about the precise values or their distribution over the lifetime of the application. As objects might be shared, names are guaranteed to be unique within a class of OpenAL objects, but no guarantees are made across different classes of objects. Objects that are unique (singletons), like the listener, do not require and do not have an integer "name".

2.12. Requesting Object Names

OpenAL provides calls to obtain object names. The application requests a number of objects of a given category using alGen{Object}s. The actual values of the names returned are implementation dependent. No guarantees on range or value are made.

Allocation of object names does not imply immediate allocation of resources or creation of objects: the implementation is free to defer this until a given object is actually used in mutator calls. The names are written at the memory location specified by the caller.

void alGenBuffers (ALsizei n, ALuint *bufferNames);

void alGenSources (ALsizei n, ALuint *sourceNames);

Requesting zero names is a legal NOP. OpenAL will respond with an AL_INVALID_VALUE error if the implementation knows that it can not store *n* names in the given array or if the implementation knows that it can not generate the requested number of objects due to non-memory resource restrictions. OpenAL will respond with an AL_OUT_OF_MEMORY error if it can not allocate the objects due to lack of available memory.

2.13. Releasing Object Names

OpenAL releases object names using alDelete{Object}s, implicitly requesting deletion of the objects associated with the names released. If one or more of the specified names is not valid, an AL_INVALID_NAME error will be recorded, and no objects will be deleted.

Once deleted, the names are no longer valid for use with any OpenAL function calls including calls to alDeleteBuffers or alDeleteSources. Any such use will cause an AL_INVALID_NAME error.

The OpenAL implementation is free to defer actual release of resources. Ideally, resources should be released as soon as possible, but no guarantees are made.

void alDeleteBuffers(ALsizei n, ALuint *bufferName);

void alDeleteSources(ALsizei n, ALuint *sourceName);

A playing source can be deleted – the source will be stopped automatically and then deleted. A buffer which is attached to a source can not be deleted.

2.14. Validating an Object Name

OpenAL provides calls to validate the name of an object. The application can verify whether an object name is valid using the alIs{Object} query. It returns AL_TRUE if the name passed to it is a valid object name, and AL_FALSE otherwise. alIs{Object} does not distinguish between invalid and deleted names.

ALboolean alIsBuffer (ALuint bufferName);

ALboolean alIsSource (ALuint sourceName);

2.15. Setting Object Attributes

Calls are provided to control the atrributes of OpenAL objects. These depend on the

actual properties of a given object category. The precise API for each category is discussed below. An OpenAL command affecting the state of a named object is usually of the form:

void al{Object}{n}{if}{v}(ALuint objectName, ALenum paramName, T values);

{Object} is the name of an OpenAL object – a “Buffer” or “Listener” for example.
{n} indicates the number of values to be passed in (if one, the value is omitted)
{if} indicates the type of the value to be passed in, “i” for integer, “f” for float.
{v} indicates that a vector of the given type will be passed in.
T is of the type indicated in the {if} and {v} fields.

The objectName parameter specifies the OpenAL object affected by this call. Use of an invalid name will cause an AL_INVALID_NAME error.

The object's attribute to be affected has to be named as paramName. OpenAL parameters applicable to one category of objects are not necessarily legal for another category of OpenAL objects. Specification of a parameter illegal for a given object will cause an AL_INVALID_OPERATION error.

Not all possible values for a type will be legal for a given objectName and parameterName. Use of an illegal value or a NULL value pointer will cause an AL_INVALID_VALUE error.

Any command that causes an error is a NOP.

In the case of unnamed (unique) objects, the (integer) objectName is omitted, as it is implied by the {Object} part of function name:

void al{Object}{n}{if}{v} (ALenum paramName, T values);

Here are some example OpenAL commands using the format rules above:

```
void  alListeneri( ALenum param, ALint value );
void  alListener3f( ALenum param, ALfloat value1, ALfloat value2, ALfloat value3 );
void  alListenerfv( ALenum param, const ALfloat* values );
void  alSourcef( ALuint sid, ALenum param, ALfloat value );
```

2.16. Querying Object Attributes

Calls to query their current attributes are provided for named and for unique OpenAL objects. These depend on the actual properties of a given object category. The performance of such queries is implementation dependent, no performance guarantees are made. The valid values for the parameter paramName are identical to the ones legal for the corresponding attribute setting function.

void alGet{Object}{n}{if}{v} (uint objectName, enum paramName, T * destination);

For unnamed unique Objects, the objectName is omitted as it is implied by the function name:

void alGet{Object}{n}{if}{v} (enum paramName, T * destination);

The definitions of {Object}, {n}, {if}, {v} and T are the same as with the set commands.

Use of an invalid name will cause an AL_INVALID_NAME error. Specification of an illegal parameter type (token) will cause an AL_INVALID_ENUM error. A call with a destination NULL pointer will be quietly ignored. The OpenAL state and destination memory will not be affected or changed by errors.

2.17. Object Attributes

Attributes affecting the processing of sounds can be set for various OpenAL object categories, or might change as an effect of OpenAL calls. The vast majority of these object properties are specific to a single OpenAL object category, but some are applicable to two or more categories and are listed separately.

The general form in which this document describes parameters is:

Name	*Signature*	*Values*	*Default*
paramName	T	Range or set	Scalar or n-tuple

Description:
The description specifies additional restrictions and details. paramName is given as the OpenAL enum defined as its name. T can be a list of legal signatures, usually the array form as well as the flat (unfolded) form.

3. State and State Requests

Most state data for OpenAL objects is retrieved or set using alGet{Object} or al{Object} calls. There is some state information that is global to the OpenAL context, such as the current error state, Doppler parameters, and the distance model.

3.1. Querying OpenAL State

3.1.1. Simple Queries

Like OpenGL, OpenAL uses a simplified interface for querying global state. The following functions accept a set of enumerations:

void alGetBooleanv(ALenum paramName, ALboolean *dest);

void alGetIntegerv(ALenum paramName, ALint *dest);

void alGetFloatv(ALenum paramName, ALfloat *dest);

void alGetDoublev(ALenum paramName, ALdouble *dest);

ALboolean alGetBoolean (ALenum paramName);

ALint alGetInteger(ALenum paramName);

ALfloat alGetFloat(ALenum paramName);

ALdouble alGetDouble(ALenum paramName);

Legal values are AL_DOPPLER_FACTOR, AL_SPEED_OF_SOUND, and AL_DISTANCE_MODEL. NULL destinations are quietly ignored. AL_INVALID_ENUM is the response to errors in specifying paramName. The amount of memory required in the destination depends on the actual state requested.

Table 3.1: Numerical Query Definitions

Name	Description
AL_DOPPLER_FACTOR	Exaggeration factor for Doppler effect
AL_SPEED_OF_SOUND	Speed of sound in same units as velocities
AL_DISTANCE_MODEL	The current distance model

3.1.2. String Queries

The application can retrieve global state information of the current OpenAL context. The

alGetString function will return a pointer to a constant string. Valid values for paramName are AL_VERSION, AL_RENDERER, AL_VENDOR, and AL_EXTENSIONS, as well as the error codes defined by OpenAL. If an invalid value for param is used, an AL_INVALID_ENUM error will be set and NULL will be returned.

const ALchar * alGetString(ALenum paramName);

Table 3.2: String Query Definitions (other than error codes)

Name	**Description**
AL_VERSION	version string in format "<spec major number>.<spec minor number> <optional vendor version information>"
AL_RENDERER	information about the specific renderer
AL_VENDOR	the name of the vendor
AL_EXTENSIONS	a list of available extensions separated by spaces

3.2. Time and Frequency

By default, OpenAL uses seconds and Hertz as units for time and frequency, respectively. A float or integral value of one for a variable that specifies quantities like duration, latency, delay, or any other parameter measured as time, specifies 1 second. For frequency, the basic unit is 1/second, or Hertz. In other words, sample frequencies and frequency cut-offs or filter parameters specifying frequencies are expressed in units of Hertz.

3.3. Space and Distance

OpenAL does not define the units of measurement for distances. The application is free to use its own units, for example, meters, inches, or parsecs. OpenAL provides means for simulating the natural attenuation of sound according to distance, and to exaggerate or reduce this effect. However, the resulting effects do not depend on the distance unit used by the application to express source and listener coordinates. OpenAL calculations are scale invariant.

The specification assumes Euclidean calculation of distances, and mandates that if two sources are sorted with respect to the Euclidean metric, the distance calculation used by the implementation has to preserve that order.

3.4. Attenuation By Distance

Samples usually use the entire dynamic range of the chosen format/encoding, independent of their real world intensity. For example, a jet engine and a clockwork both will have samples with full amplitude. The application will then have to adjust source gain accordingly to account for relative differences.

Source gain is then attenuated by distance. The effective attenuation of a source depends on many factors, among which distance attenuation and source and listener gain are only some of the contributing factors. Even if the source and listener gain exceed 1.0 (amplification beyond the guaranteed dynamic range), distance and other attenuation might ultimately limit the overall gain to a value below 1.0.

OpenAL currently supports three modes of operation with respect to distance attenuation, including one that is similar to the IASIG I3DL2 model. The application can choose one of these models (or chooses to disable distance-dependent attenuation) on a per-context basis.

void alDistanceModel(ALenum modelName);

Legal arguments are AL_NONE, AL_INVERSE_DISTANCE, AL_INVERSE_DISTANCE_CLAMPED, AL_LINEAR_DISTANCE, AL_LINEAR_DISTANCE_CLAMPED, AL_EXPONENT_DISTANCE, and AL_EXPONENT_DISTANCE_CLAMPED. AL_NONE bypasses all distance attenuation calculation for all sources. The implementation is expected to optimize this situation. AL_INVERSE_DISTANCE_CLAMPED is the IASIG I3DL2 model, with AL_REFERENCE_DISTANCE indicating both the reference distance and the distance below which gain will be clamped. AL_INVERSE_DISTANCE is equivalent to the IASIG I3DL2 model with the exception that AL_REFERENCE_DISTANCE does not imply any clamping. The linear models are not physically realistic, but do allow full attenuation of a source beyond a specified distance. The OpenAL implementation is still free to apply any range clamping as necessary. The current distance model chosen can be queried using alGetInteger{v} and AL_DISTANCE_MODEL.

With all the distance models, if the formula can not be evaluated then the source will not be attenuated. For example, if a linear model is being used with AL_REFERENCE_DISTANCE equal to AL_MAX_DISTANCE, then the gain equation will have a divide-by-zero error in it. In this case, there is no attenuation for that source.

The default attenuation model is AL_INVERSE_DISTANCE_CLAMPED.

3.4.1. Inverse Distance Rolloff Model

The following formula describes the distance attenuation defined by the Inverse Distance Attenuation Model.

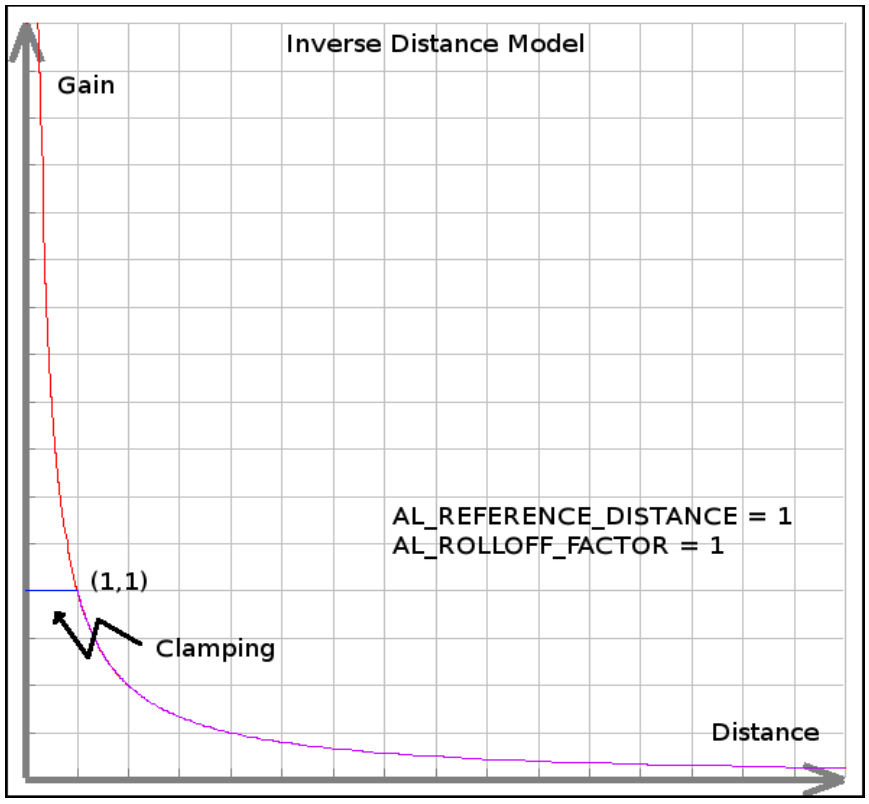
```
gain = AL_REFERENCE_DISTANCE / (AL_REFERENCE_DISTANCE +
        AL_ROLLOFF_FACTOR *
        (distance – AL_REFERENCE_DISTANCE));
```

The AL_REFERENCE_DISTANCE parameter used here is a per-source attribute that can be set and queried using the AL_REFERENCE_DISTANCE token. AL_REFERENCE_DISTANCE is the distance at which the listener will experience AL_GAIN (unless the implementation had to clamp effective AL_GAIN to the available dynamic range). AL_ROLLOFF_FACTOR is per-source parameter the application can use to increase or decrease the range of a source by decreasing or increasing the attenuation, respectively. The default value is 1.

3.4.2. Inverse Distance Clamped Model

This is the Inverse Distance Rolloff Model model, extended to guarantee that for distances below AL_REFERENCE_DISTANCE, gain is clamped. This mode is equivalent to the IASIG I3DL2 distance model.

```
distance = max(distance,AL_REFERENCE_DISTANCE);
distance = min(distance,AL_MAX_DISTANCE);
gain = AL_REFERENCE_DISTANCE / (AL_REFERENCE_DISTANCE +
            AL_ROLLOFF_FACTOR *
            (distance – AL_REFERENCE_DISTANCE));
```



3.4.3. Linear Distance Rolloff Model

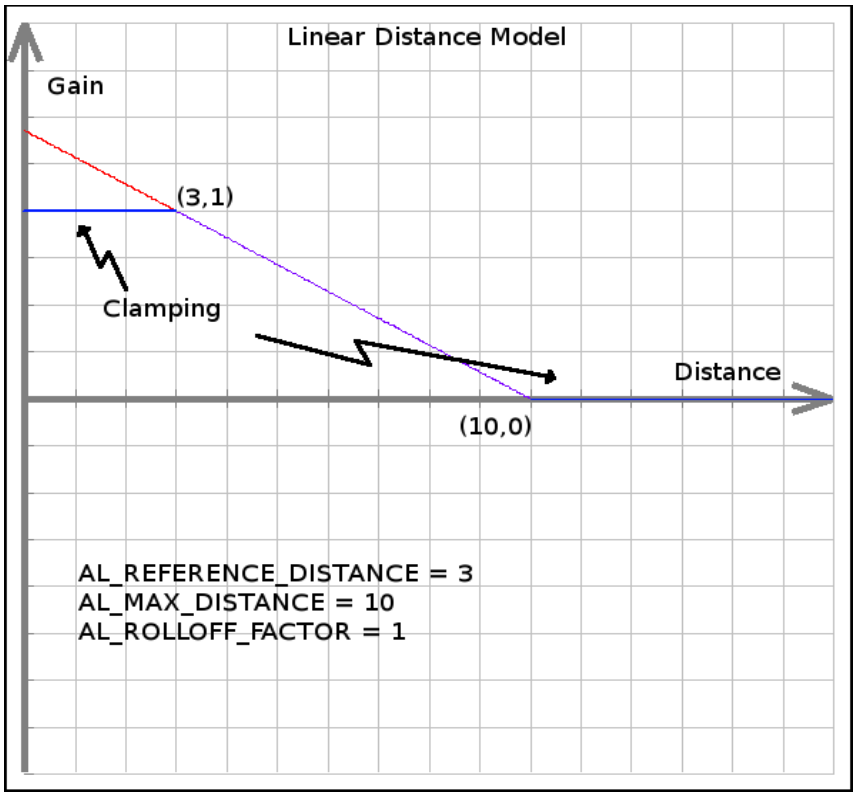
This models a linear drop-off in gain as distance increases between the source and listener.

```
distance = min(distance, AL_MAX_DISTANCE) // avoid negative gain
gain = (1 – AL_ROLLOFF_FACTOR * (distance –
                AL_REFERENCE_DISTANCE) /
                (AL_MAX_DISTANCE – AL_REFERENCE_DISTANCE))
```

3.4.4. Linear Distance Clamped Model

This is the linear model, extended to guarantee that for distances below AL_REFERENCE_DISTANCE, gain is clamped.

```
distance = max(distance, AL_REFERENCE_DISTANCE)
distance = min(distance, AL_MAX_DISTANCE)
gain = (1 – AL_ROLLOFF_FACTOR * (distance –
                AL_REFERENCE_DISTANCE) /
                (AL_MAX_DISTANCE – AL_REFERENCE_DISTANCE))
```



3.4.5. Exponential Distance Rolloff Model

This models an exponential dropoff in gain as distance increases between the source and listener.

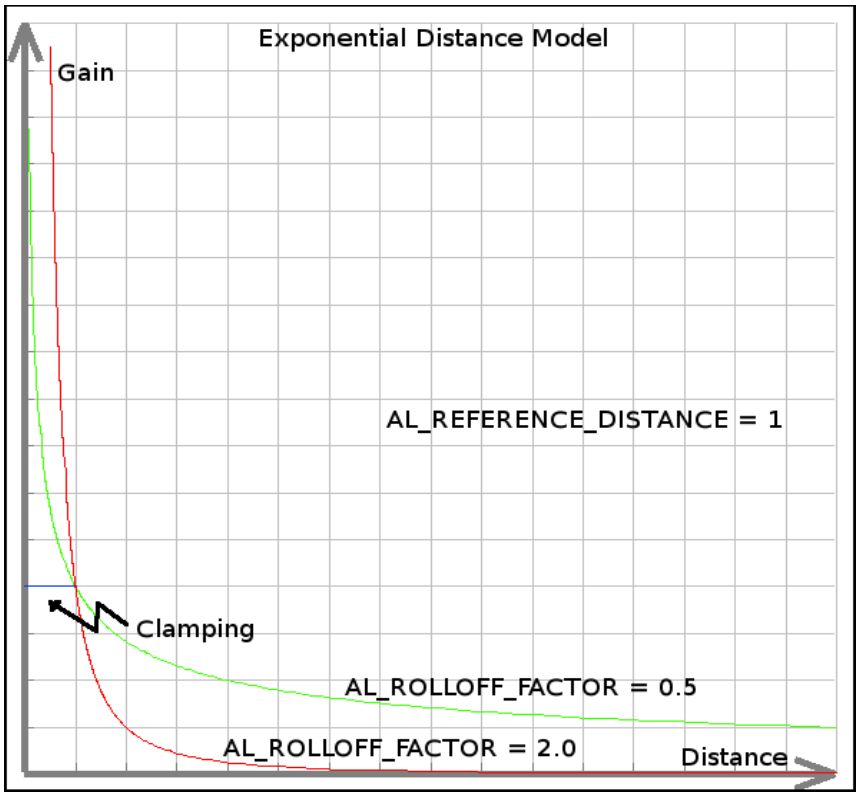
```
gain = (distance / AL_REFERENCE_DISTANCE) ^
                    (- AL_ROLLOFF_FACTOR)
```

where the ^ operation raises its first operand to the power of its second operand.

3.4.6. Exponential Distance Clamped Model

This is the exponential model, extended to guarantee that for distances below AL_REFERENCE_DISTANCE, gain is clamped.

```
distance = max(distance, AL_REFERENCE_DISTANCE)
distance = min(distance, AL_MAX_DISTANCE)
gain = (distance / AL_REFERENCE_DISTANCE) ^
                    (- AL_ROLLOFF_FACTOR)
```



3.5. Evaluation of Gain/Attenuation Related State

While amplification and attenuation commute (multiplication of scaling factors), clamping operations do not. The order in which various gain related operations are applied is:

1. Distance attenuation is calculated first, including minimum (AL_REFERENCE_DISTANCE) and maximum (AL_MAX_DISTANCE) thresholds.
2. The result is then multiplied by source gain (AL_GAIN).
3. If the source is directional (AL_CONE_INNER_ANGLE less than AL_CONE_OUTER_ANGLE), an angle-dependent attenuation is calculated depending on AL_CONE_OUTER_GAIN, and multiplied with the distance dependent attenuation. The resulting attenuation factor for the given angle and distance between listener and source is multiplied with source AL_GAIN.
4. The effective gain computed this way is compared against AL_MIN_GAIN and AL_MAX_GAIN thresholds.
5. The result is guaranteed to be clamped to [AL_MIN_GAIN, AL_MAX_GAIN], and subsequently multiplied by listener gain which serves as an overall volume control. The implementation is free to clamp listener gain if necessary due to hardware or implementation constraints.

3.5.1. No Culling By Distance

With the DirectSound3D compatible Inverse Clamped Distance Model, OpenAL provides

a per-source AL_MAX_DISTANCE attribute that can be used to define a distance beyond which the source will not be further attenuated by distance. The DS3D distance attenuation model and its clamping of volume is also extended by a mechanism to cull (mute) sources from processing, based on distance. However, the OpenAL does not support culling a source from processing based on a distance threshold.

At this time OpenAL does not support culling at all. Culling based on distance, or bounding volumes, or other criteria, is left to the application. For example, the application might employ sophisticated techniques to determine whether sources are audible. In particular, rule based culling inevitably introduces acoustic artifacts. For example, if the listener-source distance is nearly equal to the culling threshold distance, but varies above and below, there will be popping artifacts in the absence of hysteresis.

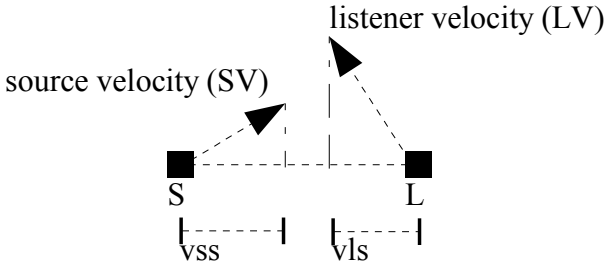
3.5.2. Velocity Dependent Doppler Effect

The Doppler Effect depends on the velocities of source and listener relative to the medium, and the propagation speed of sound in that medium. The application might want to emphasize or de-emphasize the Doppler Effect as physically accurate calculation might not give the desired results. The amount of frequency shift (pitch change) is proportional to the speed of listener and source along their line of sight.

The Doppler Effect as implemented by OpenAL is described by the formula below. Effects of the medium (air, water) moving with respect to listener and source are ignored.

SS: AL_SPEED_OF_SOUND = speed of sound (default value 343.3)
DF: AL_DOPPLER_FACTOR = Doppler factor (default 1.0)
vls: Listener velocity scalar (scalar, projected on source-to-listener vector)
vss: Source velocity scalar (scalar, projected on source-to-listener vector)
f: Frequency of sample
f': effective Doppler shifted frequency

Graphic representation of vls and vss:



3D Mathematical representation of vls and vss:

Mag(vector) = sqrt(vector.x * vector.x + vector.y * vector.y + vector.z * vector.z)

DotProduct(v1, v2) = (v1.x * v2.x + v1.y * v2.y + v1.z * v2.z)
SL = source to listener vector
SV = Source velocity vector
LV = Listener velocity vector

vls = DotProduct(SL, LV) / Mag(SL)
vss = DotProduct(SL, SV) / Mag(SL)

Dopper Calculation:

vss = min(vss, SS/DF)
vls = min(vls, SS/DF)
f' = f * (SS - DF*vls) / (SS - DF*vss)

There are two API calls global to the current context that provide control of the speed of sound and Doppler factor. AL_DOPPLER_FACTOR is a simple scaling of source and listener velocities to exaggerate or deemphasize the Doppler (pitch) shift resulting from the calculation.

void alDopplerFactor(ALfloat dopplerFactor);

A negative value will result in an AL_INVALID_VALUE error, the command is then ignored. The default value is 1. The current setting can be queried using alGetFloat{v} and AL_DOPPLER_FACTOR. The implementation is free to optimize the case of AL_DOPPLER_FACTOR being set to zero, as this effectively disables the effect.

AL_SPEED_OF_SOUND allows the application to change the reference (propagation) speed used in the Doppler calculation. The source and listener velocities should be expressed in the same units as the speed of sound.

void alSpeedOfSound(ALfloat speed);

A negative or zero value will result in an AL_INVALID_VALUE error, and the command is ignored. The default value is 343.3 (appropriate for velocity units of meters and air as the propagation medium). The current setting can be queried using alGetFloat{v} and AL_SPEED_OF_SOUND.

Distance and velocity units are completely independent of one another (so you could use different units for each if desired).

A note for OpenAL library implementors regarding OpenAL 1.0:

The OpenAL 1.1 Doppler implementation is different than that of OpenAL 1.0, because

the older implementation was confusing and not implemented consistently. The new "speed of sound" property makes the 1.1 implementation more intuitive than the old implementation. If your implementation wants to support the AL_DOPPLER_VELOCITY parameter (the alDopplerVelocity call will remain as an entry point so that 1.0 applications can link with a 1.1 library), the above formula can be changed to the following:

*vss = min(vss, (SS * DV)/DF)*
*vls = min(vls, (SS * DV)/DF)*
*f' = f * (SS * DV - DF*vls) / (SS * DV - DF*vss)*

OpenAL 1.1 programmers would never use AL_DOPPLER_VELOCITY (which defaults to 1.0).

4. Listener and Sources

4.1. Basic Listener and Source Attributes

This section introduces basic attributes that can be set both for the listener object and for source objects. The "Signature" for each attribute refers to the type or types which can be used to represent that attribute. For instance, AL_POSITION can be represented with a floating-point vector ("fv" -- alGetSourcefv or alSourcefv) or with three individual floating-point values ("3f" -- alGetSource3f or alSource3f).

The OpenAL listener object and source objects have attributes to describe their position, velocity and orientation in three dimensional space. OpenAL -- like OpenGL -- uses a right-handed Cartesian coordinate system (RHS), where in a frontal default view X (thumb) points right, Y (index finger) points up, and Z (middle finger) points towards the viewer/camera. To switch from a left handed coordinate system (LHS) to a right handed coordinate systems, flip the sign on the Z coordinate.

Name	***Signature***	***Value***	***Default***
AL_POSITION	fv, 3f, iv, 3i	Any except NaN	{0.0f, 0.0f, 0.0f}

Description:
AL_POSITION specifies the current location of the object in the world coordinate system. Any 3-tuple of valid float values is allowed. Implementation behavior on encountering NaN and infinity is not defined. The object position is always defined in the world coordinate system.

Name	***Signature***	***Values***	***Default***
AL_VELOCITY	fv, 3f, iv, 3i	Any except NaN	{0.0f, 0.0f, 0.0f}

Description:
AL_VELOCITY specifies the current velocity (speed and direction) of the object, in the world coordinate system. Any 3-tuple of valid float/double values is allowed. The object AL_VELOCITY does not affect the source's position. OpenAL does not calculate the velocity from subsequent position updates, nor does it adjust the position over time based on the specified velocity. Any such calculation is left to the application. For the purposes of sound processing, position and velocity are independent parameters affecting different aspects of the sounds.

AL_VELOCITY is taken into account by the driver to synthesize the Doppler effect perceived by the listener for each source, based on the velocity of both source and listener, and the Doppler related parameters.

Name	*Signature*	*Values*	*Default*
AL_GAIN	f, fv	[0, any]	1.0f

Description:
AL_GAIN defines a scalar amplitude multiplier. As a source attribute, it applies to that particular source only. As a listener attribute, it effectively applies to all sources in the current context. The default 1.0 means that the sound is unattenuated. An AL_GAIN value of 0.5 is equivalent to an attenuation of 6 dB. The value zero equals silence (no contribution to the output mix). Driver implementations are free to optimize this case and skip mixing and processing stages where applicable. The implementation is in charge of ensuring artifact-free (click-free) changes of gain values and is free to defer actual modification of the sound samples, within the limits of acceptable latencies.

AL_GAIN larger than one (i.e. amplification) is permitted for source and listener. However, the implementation is free to clamp the total gain (effective gain per-source multiplied by the listener gain) to one to prevent overflow.

4.2. Listener Object

The listener object defines various properties that affect processing of the sound for the actual output. The listener is unique for an OpenAL Context, and has no name. By controlling the listener, the application controls the way the user experiences the virtual world, as the listener defines the sampling/pick-up point and orientation, and other parameters that affect the output stream.

The destination output device (e.g. speakers, headphones) and the method used to render the 3D positioning (e.g. HRTFs) is implementation and hardware-dependent.

4.2.1. Listener Attributes

Several source attributes also apply to listener: AL_POSITION, AL_VELOCITY, AL_GAIN. In addition, some attributes are listener specific.

Name	*Signature*	*Values*	*Default*
AL_ORIENTATION	fv, iv	Any except NaN	{(0.0f, 0.0f, -1.0f), (0.0f, 1.0f, 0.0f)}

Description:
AL_ORIENTATION is a pair of 3-tuples consisting of an 'at' vector and an 'up' vector, where the 'at' vector represents the 'forward' direction of the listener and the orthogonal projection of the 'up' vector into the subspace perpendicular to the 'at' vector represents the 'up' direction for the listener. OpenAL expects two vectors that are linearly independent. These vectors are not expected to be normalized. If the two vectors are linearly dependent, behavior is undefined.

4.2.2. Changing Listener Attributes

Listener attributes are changed using the listener group of commands.

void alListener{n}{if}{v}(ALenum paramName, T values);

4.2.3. Querying Listener Attributes

Listener state is maintained inside the OpenAL implementation and can be queried in full. See Querying Object Attributes. The valid values for paramName are identical to the ones for the listener* command.

void alGetListener{n}{if}{v}(ALenum paramName, T *values);

4.3. Source Objects

Sources specify attributes like position, velocity, and a buffer with sample data. By controlling a source's attributes the application can modify and parameterize the static sample data provided by the buffer referenced by the source. Sources define a localized sound, and encapsulate a set of attributes applied to a sound at its origin, i.e. in the very first stage of the processing on the way to the listener. Source related effects have to be applied before listener related effects unless the output is invariant to any collapse or reversal of order. OpenAL also provides additional functions to manipulate and query the execution state of sources: the current playing status of a source (started, stopped, paused), including access to the current sampling position within the associated buffer.

4.3.1. Managing Source Names

OpenAL provides calls to request and release source names handles. Calls to control source execution state are also provided.

Requesting a Source Name

The application requests a number of sources using alGenSources.

void alGenSources(ALsizei n, ALuint *srcNames);

This call creates *n* sources, putting the source names in the srcNames array.

Releasing Source Names

The application requests deletion of a number of sources by alDeleteSources.

void alDeleteSources(ALsizei n, ALuint *sources);

A playing source can be deleted – the source will be stopped automatically and then deleted.

Validating a Source Name

The application can verify whether a source name is valid using the alIsSource query.

ALboolean alIsSource (ALuint sourceName);

4.3.2. Source Attributes

This section lists the attributes that are set per-source, affecting the processing of the current buffer. Some of these attributes can also be set for buffer queue entries.

Source Positioning

Name	*Signature*	*Values*	*Default*
AL_SOURCE_RELATIVE	i, iv	AL_TRUE, AL_FALSE	AL_FALSE

Description:

AL_SOURCE_RELATIVE set to AL_TRUE indicates that the position, velocity, cone, and direction properties of a source are to be interpreted relative to the listener position.

Source Type

Name	*Signature*	*Values*	*Default*
AL_SOURCE_TYPE	i, iv	AL_UNDETERMINED, AL_STATIC, AL_STREAMING	AL_UNDETERMINED

Description:
AL_SOURCE_TYPE is a read-only property indicating whether a source is ready to queue buffers, ready to use a static buffer, or is in an undetermined state where it can be used for either streaming or static playback.

When first created, a source will be in the AL_UNDETERMINED state. If a buffer is then attached using alSourcei(sid, AL_BUFFER, bid), then the source will enter the AL_STATIC state. If the first buffer attached to a source is attached using alSourceQueueBuffers, then the source will enter the AL_STREAMING state. Attaching the NULL buffer using alSourcei(sid, AL_BUFFER, NULL) to a source of either type will reset the state to AL_UNDETERMINED, and attaching any buffer to a streaming source will change the state to AL_STATIC.

Attempting to queue a buffer on a static source will result in an AL_INVALID_OPERATION error.

Buffer Looping

Name	*Signature*	*Values*	*Default*
AL_LOOPING	i, iv	AL_TRUE, AL_FALSE	AL_FALSE

Description:
AL_LOOPING is a flag that indicates that the source will not be in AL_STOPPED state once it reaches the end of last buffer in the buffer queue. Instead, the source will immediately promote to AL_INITIAL and AL_PLAYING. The default value is AL_FALSE. AL_LOOPING can be changed on a source in any execution state. In particular, it can be changed on a AL_PLAYING source.

Current Buffer

Name	*Signature*	*Values*	*Default*
AL_BUFFER	i, iv	any valid bufferName	AL_NONE

Description:
Specifies the current buffer object, making it the head entry in the source's queue. Using AL_BUFFER on a source in the AL_STOPPED or AL_INITIAL state empties the entire queue, then appends the one buffer specified.

For a source in the AL_PLAYING or AL_PAUSED state, setting AL_BUFFER will result in the AL_INVALID_OPERATION error state being set. AL_BUFFER can be applied only to sources in the AL_INITIAL and AL_STOPPED states. Specifying an invalid buffer name (either because the buffer name does not exist or because that buffer can't be attached to the specified source) will result in an AL_INVALID_VALUE error while specifying an invalid source name results in an AL_INVALID_NAME error.

AL_NONE (NULL or 0), is a valid buffer name. alSourcei(sName, AL_BUFFER, AL_NONE) is a legal way to release the current buffer queue on a source in the AL_INITIAL or AL_STOPPED state, whether the source has just one entry (current buffer) or more. The alSourcei(sName, AL_BUFFER, AL_NONE) call still causes an AL_INVALID_OPERATION for any source in the AL_PLAYING or AL_PAUSED state, consequently it cannot be used to mute or stop a source.

Queue State Queries

Name	*Signature*	*Values*	*Default*
AL_BUFFERS_QUEUED	i, iv	[0, any]	none

Query only. Query the number of buffers in the queue of a given source. This includes those not yet played, the one currently playing, and the ones that have been played already. This will return 0 if the current and only buffer name is 0.

Name	*Signature*	*Values*	*Default*
AL_BUFFERS_PROCESSED	i, iv	[0, any]	none

Query only. Query the number of buffers that have been played by a given source. Indirectly, this gives the index of the buffer currently playing. Used to determine how many slots are needed for unqueuing them. On a source in the AL_STOPPED state, all buffers are processed. On a source in the AL_INITIAL state, no buffers are processed, all buffers are pending. This will return 0 if the current and only buffer name is 0.

Bounds on Gain

Name	*Signature*	*Values*	*Default*
AL_MIN_GAIN	f, fv	[0.0f, 1.0f]	0.0f

Description:
AL_MIN_GAIN is a scalar amplitude threshold. It indicates the minimal AL_GAIN that is always guaranteed for this source. At the end of the processing of various attenuation factors such as distance based attenuation and source AL_GAIN, the effective gain calculated is compared to this value. If the effective gain is lower than AL_MIN_GAIN, AL_MIN_GAIN is applied. This happens before the listener gain is applied. If a zero AL_MIN_GAIN is set, then the effective gain will not be corrected.

Name	*Signature*	*Values*	*Default*
AL_MAX_GAIN	f, fv	[0.0f, 1.0f]	1.0f

Description:
AL_MAX_GAIN defines a scalar amplitude threshold. It indicates the maximal AL_GAIN permitted for this source. At the end of the processing of various attenuation factors such as distance based attenuation and source AL_GAIN, the effective gain calculated is compared to this value. If the effective gain is higher than AL_MAX_GAIN, AL_MAX_GAIN is applied. This happens before the listener AL_GAIN is applied. If the listener gain times AL_MAX_GAIN still exceeds the maximum gain the implementation can handle, the implementation is free to clamp. If a zero AL_MAX_GAIN is set, then the source is effectively muted. The implementation is free to optimize for this situation, but no optimization is required or recommended as setting GAIN to zero is the proper way to mute a source.

Distance Model Attributes

Table 4.12: REFERENCE_DISTANCE Attribute

Name	*Signature*	*Values*	*Default*
AL_REFERENCE_DISTANCE	f, fv, i, iv	[0, any]	1.0f

This is used for distance attenuation calculations based on inverse distance with rolloff. Depending on the distance model it will also act as a distance threshold below which gain is clamped. See the section on distance models for details.

Table 4.13: ROLLOFF_FACTOR Attribute

Name	*Signature*	*Values*	*Default*
AL_ROLLOFF_FACTOR	f, fv, i, iv	[0, any]	1.0f

This is used for distance attenuation calculations based on inverse distance with rolloff. For distances smaller than AL_MAX_DISTANCE (and, depending on the distance model, larger than AL_REFERENCE_DISTANCE), this will scale the distance attenuation over the applicable range. See section on distance models for details how the attenuation is computed as a function of the distance.

In particular, AL_ROLLOFF_FACTOR can be set to zero for those sources that are supposed to be exempt from distance attenuation. The implementation is encouraged to optimize this case, bypassing distance attenuation calculation entirely on a per-source basis.

Table 4.14: MAX_DISTANCE Attribute

Name	*Signature*	*Values*	*Default*
AL_MAX_DISTANCE	f, fv, i, iv	[0, any]	MAX_FLOAT

This is used for distance attenuation calculations based on inverse distance with rolloff, if the Inverse Clamped Distance Model is used. In this case, distances greater than AL_MAX_DISTANCE will be clamped to AL_MAX_DISTANCE. AL_MAX_DISTANCE based clamping is applied before AL_MIN_GAIN clamping, so if the effective gain at AL_MAX_DISTANCE is larger than AL_MIN_GAIN, AL_MIN_GAIN will have no effect. No culling is supported.

Frequency Shift by Pitch

Table 4.15: SOURCE_PITCH Attribute

Name	*Signature*	*Values*	*Default*
AL_PITCH	f, fv	(0.0f, any]	1.0f

Description:

Desired pitch shift, where 1.0 equals identity. Each reduction by 50 percent equals a pitch shift of -12 semitones (one octave reduction). Each doubling equals a pitch shift of 12 semitones (one octave increase). Zero is not a legal value. Implementations may clamp the actual output pitch range to any values subject to the implementation's own limits.

Direction and Cone

Each source can be directional, depending on the settings for AL_CONE_INNER_ANGLE and AL_CONE_OUTER_ANGLE. There are three zones defined: the inner cone, the outside zone, and the transitional zone in-between. The angle-dependent gain for a directional source is constant inside the inner cone, and changes over the transitional zone to the value specified outside the outer cone. Source AL_GAIN is applied for the inner cone, with an application selectable AL_CONE_OUTER_GAIN factor to define the gain in the outer zone. In the transitional zone linear interpolation between AL_GAIN and AL_GAIN times AL_CONE_OUTER_GAIN is applied.

Table 4.16: Source DIRECTION Attribute

Name	*Signature*	*Values*	*Default*
AL_DIRECTION	fv, 3f, iv, 3i	Any except NaN	(0.0f, 0.0f, 0.0f)

Description:
If AL_DIRECTION does not equal the zero vector, the source is directional. The sound emission is presumed to be symmetric around the direction vector (cylinder symmetry). sources are not oriented in full 3 degrees of freedom, only two angles are effectively needed.

The zero vector is default, indicating that a source is not directional. Specifying a non-zero vector will make the source directional. Specifying a zero vector for a directional source will effectively mark it as non-directional.

Table 4.17: Source CONE_INNER_ANGLE Attribute

Name	*Signature*	*Values*	*Default*
AL_CONE_INNER_ANGLE	f, fv, i, iv	Any except NaN	360.0f

Description:
Inside angle of the sound cone, in degrees. The default of 360 means that the inner angle covers the entire world, which is equivalent to an omni-directional source.

Table 4.18: Source CONE_OUTER_ANGLE Attribute

Name	*Signature*	*Values*	*Default*
AL_CONE_OUTER_ANGLE	f, fv, i, iv	Any except NaN	360.0f

Description: Outer angle of the sound cone, in degrees. The default of 360 means that the outer angle covers the entire world. If the inner angle is also 360, then the zone for angle-dependent attenuation is zero.

Table 4.19: Source CONE_OUTER_GAIN Attribute

Name	*Signature*	*Values*	*Default*
AL_CONE_OUTER_GAIN	f, fv	[0.0f, 1.0f]	0.0f

Description: the factor with which AL_GAIN is multiplied to determine the effective gain outside the cone defined by the outer angle. The effective gain applied outside the outer cone is AL_GAIN times AL_CONE_OUTER_GAIN. Changing AL_GAIN affects all directions, i.e. the source is attenuated in all directions, for any position of the listener. The application has to change AL_CONE_OUTER_GAIN as well if a different behavior is desired.

Offset

Table 4.20: Source AL_SEC_OFFSET Attribute

Name	*Signature*	*Values*	*Default*
AL_SEC_OFFSET	f, fv, i, iv	[0.0f, any]	N/A

Description: the playback position, expressed in seconds (the value will loop back to zero for looping sources).

When setting AL_SEC_OFFSET on a source which is already playing, the playback will jump to the new offset unless the new offset is out of range, in which case an AL_INVALID_VALUE error is set. If the source is not playing, then the offset will be applied on the next alSourcePlay call.

The position is relative to the beginning of all the queued buffers for the source, and any queued buffers traversed by a set call will be marked as processed.

This value is based on byte position, so a pitch-shifted source will have an exaggerated playback speed. For example, you can be 0.500 seconds into a buffer having taken only 0.250 seconds to get there if the pitch is set to 2.0.

Table 4.21: Source AL_SAMPLE_OFFSET Attribute

Name	*Signature*	*Values*	*Default*
AL_SAMPLE_OFFSET	f, fv, i, iv	[0.0f, any]	N/A

Description: the playback position, expressed in samples (the value will loop back to zero for looping sources). For a compressed format, this value will represent an exact offset within the uncompressed data.

When setting AL_SAMPLE_OFFSET on a source which is already playing, the playback will jump to the new offset unless the new offset is out of range, in which case an AL_INVALID_VALUE error is set. If the source is not playing, then the offset will be applied on the next alSourcePlay call. An alSourceStop, alSourceRewind, or a second alSourcePlay call will reset the offset to the beginning of the buffer.

The position is relative to the beginning of all the queued buffers for the source, and any queued buffers traversed by a set call will be marked as processed.

Table 4.22: Source AL_BYTE_OFFSET Attribute

Name	***Signature***	***Values***	***Default***
AL_BYTE_OFFSET	f, fv, i, iv	[0.0f, any]	N/A

Description: the playback position, expressed in bytes (the value will loop back to zero for looping sources). For a compressed format, this value may represent an approximate offset within the compressed data buffer.

When setting AL_BYTE_OFFSET on a source which is already playing, the playback will jump to the new offset unless the new offset is out of range, in which case an AL_INVALID_VALUE error is set. If the source is not playing, then the offset will be applied on the next alSourcePlay call. An alSourceStop, alSourceRewind, or a second alSourcePlay call will reset the offset to the beginning of the buffer.

The position is relative to the beginning of all the queued buffers for the source, and any queued buffers traversed by a set call will be marked as processed.

4.3.3. Changing Source Attributes

The source specifies the position and other properties as taken into account during sound processing.

void alSource{n}{if} (ALuint sourceName, ALenum paramName, T value);

void alSource{n}{if}v (ALuint sourceName, ALenum paramName, T *values);

4.3.4. Querying Source Attributes

Source state is maintained inside the OpenAL implementation, and the current attributes can be queried. The performance of such queries is implementation dependent, no performance guarantees are made. The valid values for the paramName parameter are identical to the ones for the source set calls.

void alGetSource{n}{if}{v} (ALuint sourceName, ALenum paramName, T *values);

4.3.5. Queuing Buffers with a Source

OpenAL does not specify a built-in streaming mechanism. There is no mechanism to stream data into a buffer object. Instead, the API has a more flexible and versatile mechanism to queue buffers for sources.

There are many ways to use this feature, with streaming being only one of them.

Streaming is replaced by queuing static buffers. This effectively moves any multi-buffer caching into the application and allows the application to select how many buffers it wants to use, the size of the buffers, and whether these are re-used in cycle, pooled, or thrown away.

Looping (over a finite number of repetitions) can be implemented by explicitly repeating buffers in the queue. Infinite loops can (theoretically) be accomplished by sufficiently large repetition counters. If only a single buffer is supposed to be repeated infinitely, using the respective source attribute is recommended.

Loop Points for restricted looping inside a buffer can in many cases be replaced by splitting the sample into several buffers and queuing the sample fragments (including repetitions) accordingly.

Buffers can be queued, unqueued after they have been used, and either be deleted, or refilled and queued again. Splitting large samples over several buffers maintained in a queue has a distinct advantages over approaches that require explicit management of samples and sample indices.

Queuing Command

The application can queue up one or multiple buffer names using alSourceQueueBuffers. The buffers will be queued in the sequence in which they appear in the array.

```
void alSourceQueueBuffers (ALuint sourceName, ALsizei numBuffers,
        ALuint *bufferNames);
```

This command is legal on a source in any playback state (to allow for streaming, queuing has to be possible on a AL_PLAYING source).

All buffers in a queue must have the same format and attributes, with the exception of the NULL buffer (i.e., 0) which can always be queued. An attempt to mix formats or other buffer attributes will result in a failure and an AL_INVALID_VALUE error will be thrown. If the queue operation fails, the source queue will remain unchanged (even if some of the buffers could have been queued).

Unqueuing Command

Once a queue entry for a buffer has been appended to a queue and is pending processing, it should not be changed. Removal of a given queue entry is not possible unless either the source is stopped (in which case then entire queue is considered processed), or if the queue entry has already been processed (AL_PLAYING or AL_PAUSED source). A playing source will enter the AL_STOPPED state if it completes playback of the last buffer in its queue (the same behavior as when a single buffer has been attached to a source and has finished playback).

The alSourceUnqueueBuffers command removes a number of buffers entries that have finished processing, in the order of appearance, from the queue. The operation will fail with an AL_INVALID_VALUE error if more buffers are requested than available, leaving the destination arguments unchanged.

void alSourceUnqueueBuffers (ALuint sourceName, ALsizei numEntries,
    ALuint *bufferNames);

4.3.6. Managing Source Execution

The execution state of a source can be queried. OpenAL provides a set of functions that initiate state transitions causing sources to start and stop execution.

Source State Query

The application can query the current state of any source using alGetSource with the parameter name AL_SOURCE_STATE. Each source can be in one of four possible execution states: AL_INITIAL, AL_PLAYING, AL_PAUSED, AL_STOPPED. Sources that are either AL_PLAYING or AL_PAUSED are considered active. Sources that are AL_STOPPED or AL_INITIAL are considered inactive. Only AL_PLAYING sources are included in the processing. The implementation is free to skip those processing stages for sources that have no effect on the output (e.g. mixing for a source muted by zero GAIN, but not sample offset increments). Depending on the current state of a source certain (e.g. repeated) state transition commands are legal NOPs: they will be ignored, no error is generated.

State Transition Commands

The default state of any source is INITIAL. From this state it can be propagated to any other state by appropriate use of the commands below. There are no irreversible state transitions.

void alSourcePlay (ALuint sName);

void alSourcePause (ALuint sName);

void alSourceStop (ALuint sName);

void alSourceRewind (ALuint sName);

The functions are also available as a vector variant, which guarantees synchronized operation on a set of sources.

void alSourcePlayv (ALsizei n, const ALuint * sNames);

void alSourcePausev (ALsizei n, cost ALuint *sNames);

void alSourceStopv (ALsizei n, const ALuint *sNames);

void alSourceRewindv (ALsizei n, const ALuint *sNames);

The following state/command/state transitions are defined:

- alSourcePlay applied to an AL_INITIAL source will promote the source to AL_PLAYING, thus the data found in the buffer will be fed into the processing, starting at the beginning. alSourcePlay applied to a AL_PLAYING source will restart the source from the beginning. It will not affect the configuration, and will leave the source in AL_PLAYING state, but reset the sampling offset to the beginning. alSourcePlay applied to a AL_PAUSED source will resume processing using the source state as preserved at the alSourcePause operation. alSourcePlay applied to a AL_STOPPED source will propagate it to AL_INITIAL then to AL_PLAYING immediately.
- alSourcePause applied to an AL_INITIAL source is a legal NOP. alSourcePause applied to a AL_PLAYING source will change its state to AL_PAUSED. The source is exempt from processing, its current state is preserved. alSourcePause applied to a AL_PAUSED source is a legal NOP. alSourcePause applied to a AL_STOPPED source is a legal NOP.
- alSourceStop applied to an AL_INITIAL source is a legal NOP. alSourceStop applied to a AL_PLAYING source will change its state to AL_STOPPED. The source is exempt from processing, its current state is preserved. alSourceStop applied to a AL_PAUSED source will change its state to AL_STOPPED, with the same consequences as on a AL_PLAYING source. alSourceStop applied to a AL_STOPPED source is a legal NOP.
- alSourceRewind applied to an AL_INITIAL source is a legal NOP. alSourceRewind applied to a AL_PLAYING source will change its state to AL_STOPPED then AL_INITIAL. The source is exempt from processing: its current state is preserved, with the exception of the sampling offset, which is reset to the beginning. alSourceRewind applied to a AL_PAUSED source will change its state to AL_INITIAL, with the same consequences as on a AL_PLAYING source. alSourceRewind applied to an AL_STOPPED source promotes the source to AL_INITIAL, resetting the sampling offset to the beginning.

Resetting Configuration

Promoting a source to the AL_INITIAL state using alSourceRewind will not reset the source's properties. AL_INITIAL merely indicates that the source can be executed using the alSourcePlay command. An AL_STOPPED or AL_INITIAL source can be reset into the default configuration by using a sequence of source commands as necessary. As the application has to specify all relevant state anyway to create a useful source configuration, no reset command is provided.

5. Buffers

A buffer encapsulates OpenAL state related to storing sample data. The application can request and release buffer objects, and fill them with data. Data can be supplied compressed and encoded as long as the format is supported. Buffers can, internally, contain waveform data as uncompressed or compressed samples.

Unlike source and listener objects, buffer objects can be shared among AL contexts. Buffers are referenced by sources. A single buffer can be referred to by multiple sources. This separation allows drivers and hardware to optimize storage and processing where applicable.

The simplest supported format for buffer data is PCM. PCM data is assumed to use the processor's native byte order. Other formats use the byte order native to that format.

5.1. Buffer States

At this time, buffer states are defined for purposes of discussion. The states described in this section are not exposed through the API (can not be queried, or be set directly), and the state description used in the implementation might differ from this.

A buffer is considered to be in one of the following states, with respect to all sources:

- UNUSED: the buffer is not included in any queue for any source. In particular, the buffer is neither pending nor current for any source. The buffer name can be deleted at this time.
- PROCESSED: the buffer is listed in the queue of at least one source, but is neither pending nor current for any source. The buffer can be deleted as soon as it has been unqueued for all sources it is queued with.
- PENDING: there is at least one source for which the buffer has been queued, for which the buffer data has not yet been dereferenced. The buffer can only be unqueued for those sources that have dereferenced the data in the buffer in its entirety, and cannot be deleted or changed.

The buffer state is dependent on the state of all sources that is has been queued for. A single queue occurrence of a buffer propagates the buffer state (over all sources) from UNUSED to PROCESSED or higher. Sources that are in the AL_STOPPED or AL_INITIAL states still have queue entries that cause buffers to be PROCESSED.

A single queue entry with a single source for which the buffer is not yet PROCESSED propagates the buffer's queuing state to PENDING.

Buffers that are PROCESSED for a given source can be unqueued from that source's queue. Buffers that have been unqueued from all sources are UNUSED. Buffers that are UNUSED can be deleted, or changed by alBufferData commands.

5.2. Managing Buffer Names

OpenAL provides calls to obtain buffer names, to request deletion of a buffer object associated with a valid buffer name, and to validate a buffer name. Calls to control buffer attributes are also provided.

5.2.1. Requesting Buffers Names

The application requests a number of buffers using alGenBuffers.

void alGenBuffers (ALsizei n, ALuint *bufferNames);

This can be called at any time and multiple calls will generate multiple sets of buffers.

5.2.2. Releasing Buffer Names

The application requests deletion of a number of buffers by calling alDeleteBuffers.

Once deleted, names are no longer valid for use with AL function calls. Any such use will cause an AL_INVALID_NAME error. The implementation is free to defer actual release of resources.

void alDeleteBuffers(ALsizei n, const ALuint *bufferNames);

alIsBuffer(bname) can be used to verify deletion of a buffer. Deleting buffer name 0 is a legal NOP. A buffer which is attached to a source can not be deleted.

5.2.3. Validating a Buffer Name

The application can verify whether a buffer name is valid using the alIsBuffer query.

boolean alIsBuffer (ALuint bufferName);

5.3. Manipulating Buffer Attributes

5.3.1. Buffer Attributes

This section lists the buffer attributes that can queried. Note that the listed attributes are set using alBufferData.

Querying the attributes of a buffer with a buffer name that is not valid throws an AL_INVALID_OPERATION. Passing in an attribute name that is invalid throws an AL_INVALID_VALUE error.

Table 5.1: Buffer FREQUENCY Attribute

Name	*Signature*	*Values*	*Default*
AL_FREQUENCY	i, iv	(0, any]	0

Description: Frequency, specified in samples per second, i.e. units of Hertz [Hz]. Query by alGetBuffer. The frequency state of a buffer is set by alBufferData calls.

Table 5.2: Buffer SIZE Attribute

Name	*Signature*	*Values*	*Default*
AL_SIZE	i, iv	[0, MAX_UINT]	0

Description: Size in bytes of the buffer data. Query through alGetBuffer, can be set only using alBufferData calls. Setting an AL_SIZE of 0 is a legal NOP.

Table 5.3: Buffer BITS Attribute

Name	*Signature*	*Values*	*Default*
AL_BITS	i, iv	8, 16	16

Description: The number of bits per sample for the data contained in the buffer. Query by alGetBuffer. The number of bits is set by alBufferData calls.

Table 5.4: Buffer CHANNELS Attribute

Name	*Signature*	*Values*	*Default*
AL_CHANNELS	i, iv	1, 2	1

Description: The number of channels for the data contained in the buffer. Query by alGetBuffer. The number of channels is set by alBufferData calls.

5.3.2. Changing Buffer Attributes

A buffer-related extension may wish to set its own buffer attributes using one of the following calls:

void alBuffer{n}{if} (ALuint sourceName, ALenum paramName, T value);

void alBuffer{n}{if}v (ALuint sourceName, ALenum paramName, T *values);

5.3.3. Querying Buffer Attributes

Buffer state is maintained inside the OpenAL implementation and can be queried in full. The valid values for paramName are AL_FREQUENCY, AL_SIZE, AL_BITS, and AL_CHANNELS, and the values returned will represent the internal representation of the buffer data.

void alGetBuffer{n}{if}{v} (ALuint bufferName, ALenum paramName, T *values);

5.3.4. Specifying Buffer Content

A special case of buffer state is the actual sound sample data stored in association with the buffer. Applications can specify sample data using alBufferData.

void alBufferData (ALuint bufferName, ALenum format,
const ALvoid *data, ALsizei size, ALsizei frequency);

The data specified is copied to an internal software, or if possible, hardware buffer. The implementation is free to apply decompression, conversion, resampling, and filtering as needed. Valid formats are AL_FORMAT_MONO8, AL_FORMAT_MONO16, AL_FORMAT_STEREO8, and AL_FORMAT_STEREO16. An implementation may expose other formats through extensions.

8-bit data is expressed as an unsigned value over the range 0 to 255, 128 being an audio output level of zero.

16-bit data is expressed as a signed value over the range -32768 to 32767, 0 being an audio output level of zero. Byte order for 16-bit values is determined by the native format of the CPU.

Stereo data is expressed in an interleaved format, left channel sample followed by the right channel sample.

Buffers containing audio data with more than one channel will be played without 3D spatialization features – these formats are normally used for background music.

The size given is the number of bytes, and must be logical for the format given – an odd value for 16-bit data will always be an error, for example. An invalid size will result in an AL_INVALID_VALUE error.

Applications should always check for an error condition after attempting to specify buffer data in case an implementation has to generate an AL_OUT_OF_MEMORY or conversion related AL_INVALID_VALUE error. The application is free to reuse the memory specified by the data pointer once the call to alBufferData returns. The implementation has to dereference, e.g. copy, the data during alBufferData execution.

6. AL Contexts and the ALC API

This section of the OpenAL specification describes ALC, the OpenAL Context API. ALC is a portable API for managing OpenAL contexts, including resource sharing, locking, and unlocking. Within the core AL API the existence of a Context is implied, but the Context is not exposed. The Context encapsulates the state of a given instance of the OpenAL state machine.

The Context API makes use of ALC types which are defined separately from the OpenAL types – there is an ALCboolean, ALCchar, etc.

6.1. Managing Devices

ALC introduces the notion of a Device. A Device can be, depending on the implementation, a hardware device, or a daemon/OS service/actual server. This mechanism also permits different drivers (and hardware) to coexist within the same system, as well as allowing several applications to share system resources for audio, including a single hardware output device. The details are left to the implementation, which has to map the available backends to unique device specifiers.

6.1.1. Connecting to a Device

The alcOpenDevice function allows the application (i.e. the client program) to connect to a device (i.e. the server).

ALCdevice * alcOpenDevice (const ALCchar *deviceSpecifier);

If the function returns NULL, then no sound driver/device has been found. The argument is a null terminated string that requests a certain device or device configuration. If NULL is specified, the implementation will provide an implementation specific default.

6.1.2. Disconnecting from a Device

The alcCloseDevice function allows the application (i.e. the client program) to disconnect from a device (i.e. the server).

ALCboolean alcCloseDevice (ALCdevice *deviceHandle);

The return code will be ALC_TRUE or ALC_FALSE, indicating success or failure. Failure will occur if all the device's contexts and buffers have not been destroyed. Once closed, the deviceHandle is invalid.

6.2. Managing Rendering Contexts

All operations of the OpenAL core API affect a current context. Within the scope of OpenAL, the context is implied - it is not visible as a handle or function parameter. Only one OpenAL context per process can be current at a time. Applications maintaining

multiple OpenAL contexts, whether threaded or not, have to set the current context accordingly. Applications can have multiple threads that share one more or contexts. In other words, AL and ALC are threadsafe.

6.2.1. Context Attributes

The application can choose to specify certain attributes for a context at context-creation time. Attributes not specified explicitly are set to implementation dependent defaults.

Table 6.1: Context Creation Attributes

Name	***Description***
ALC_FREQUENCY	Frequency for mixing output buffer, in units of Hz
ALC_REFRESH	Refresh intervals, in units of Hz
ALC_SYNC	Flag, indicating a synchronous context
ALC_MONO_SOURCES	A hint indicating how many sources should be capable of supporting mono data
ALC_STEREO_SOURCES	A hint indicating how many sources should be capable of supporting stereo data

6.2.2. Creating a Context

A context is created using alcCreateContext. The device parameter has to be a valid device. The attribute list can be NULL, or a zero terminated list of integer pairs composed of valid ALC attribute tokens and requested values.

```
ALCcontext * alcCreateContext (const ALCdevice * deviceHandle,
        const ALCint * attrList);
```

Context creation will fail in the following cases:
a) if the application requests attributes that, by themselves, can not be provided
b) if the combination of specified attributes can not be provided
c) if a specified attribute, or the combination of attributes, does not match the default values for unspecified attributes

If context creation fails, the context pointer returned will be NULL.

6.2.3. Selecting a Context for Operation

To make a Context current with respect to OpenAL operation, alcMakeContextCurrent is used. The context parameter can be NULL or a valid context pointer. Using NULL results in no context being current, which is useful when shutting OpenAL down. The operation will apply to the device that the context was created for.

ALCboolean alcMakeContextCurrent (ALCcontext * context);

The return value (ALC_TRUE or ALC_FALSE) will reflect whether or not an error occurred in the call. Standard error conditions are also set during execution of this call (ALC_INVALID_CONTEXT for an invalid context pointer, for instance).

For each OS process (usually this means for each application), only one context can be current at any given time. All AL commands apply to the current context. Commands that affect objects shared among contexts (e.g. buffers) have side effects on other contexts.

6.2.4. Initiate Context Processing

The current context is the only context accessible to state changes by AL commands (aside from state changes affecting shared objects). However, multiple contexts can be processed at the same time. To indicate that a context should be processed (i.e. that internal execution state such as the offset increments are to be performed), the application uses alcProcessContext.

void alcProcessContext (ALCcontext *context);

Repeated calls to alcProcessContext are legal, and do not affect a context that is already marked as processing. The default state of a context created by alcCreateContext is that it is processing.

6.2.5. Suspend Context Processing

The application can suspend any context from processing (including the current one). To indicate that a context should be suspended from processing (i.e. that internal execution state such as offset increments are not to be changed), the application uses alcSuspendContext.

void alcSuspendContext (ALCcontext *context);

Repeated calls to alcSuspendContext are legal, and do not affect a context that is already marked as suspended.

6.2.6. Destroying a Context

void alcDestroyContext(ALCcontext * context);

The correct way to destroy a context is to first release it using alcMakeCurrent with a NULL context. Applications should not attempt to destroy a current context – doing so will not work and will result in an ALC_INVALID_OPERATION error. All sources within a context will automatically be deleted during context destruction.

6.3. ALC Queries

6.3.1. Query for Current Context

The application can query for, and obtain an handle to, the current context for the application. If there is no current context, NULL is returned.

ALCcontext * alcGetCurrentContext(void);

6.3.2. Query for a Context's Device

The application can query for, and obtain an handle to, the device of a given context.

ALCdevice * alcGetContextsDevice(ALCcontext * context);

6.3.3. Query For Extensions

To verify that a given extension is available for the current context and the device it is associated with, use

ALCboolean alcIsExtensionPresent(const ALCdevice *deviceHandle,
    const ALCchar *extName);

Invalid and unsupported string tokens return ALC_FALSE. A NULL deviceHandle is acceptable. A NULL extName will result in an ALC_INVALID_VALUE error and the return code will be ALC_FALSE. extName is not case sensitive – the implementation will convert the name to all upper-case internally (and will express extension names in upper-case).

6.3.4. Query for Function Entry Addresses

The application is expected to verify the applicability of an extension or core function entry point before requesting it by name, by use of alcIsExtensionPresent. Extension entry points can be retrieved using alcGetProcAddress.

void * alcGetProcAddress (const ALCdevice *deviceHandle, const ALchar *funcName);

Entry points can be device specific, but are not context specific. Using a NULL device handle does not guarantee that the entry point is returned, even if available for one of the available devices. Specifying a NULL name parameter will cause an ALC_INVALID_VALUE error.

6.3.5. Retrieving Enumeration Values

Enumeration/token values are device independent, but tokens defined for extensions might not be present for a given device. Using a NULL handle is legal, but only the tokens defined by the OpenAL core are guaranteed. Availability of extension tokens

depends on the ALC extension.

ALCenum alcGetEnumValue(const ALCdevice *deviceHandle, const ALCchar *enumName);

Specifying a NULL enumName parameter will cause an ALC_INVALID_VALUE error and a return value of AL_NONE.

6.3.6. Query for Error Conditions

ALC uses the same conventions and mechanisms as AL for error handling. In particular, ALC does not use conventions derived from X11 (GLX) or Windows (WGL). The alcGetError function can be used to query ALC errors.

ALCenum alcGetError(ALCdevice * deviceHandle);

Error conditions are specific to the device, and (like AL) a call to alcGetError resets the error state.

Table 6.2: Error Conditions

Name	***Description***
ALC_NO_ERROR	There is no current error.
ALC_INVALID_DEVICE	The device handle or specifier names an accessible driver/server.
ALC_INVALID_CONTEXT	The Context argument does not name a valid context.
ALC_INVALID_ENUM	A token used is not valid, or not applicable.
ALC_INVALID_VALUE	A value (e.g. Attribute) is not valid, or not applicable.
ALC_OUT_OF_MEMORY	Unable to allocate memory.

6.3.7. String Query

The application can obtain certain strings from ALC.

const ALCchar * alcGetString(ALCdevice * deviceHandle, ALCenum token);

Specifying NULL for deviceHandle when asking for ALC_EXTENSIONS will generate an ALC_INVALID_DEVICE error. The deviceHandle value is ignored when asking for ALC_DEFAULT_DEVICE_SPECIFIER or ALC_CAPTURE_DEFAULT_DEVICE_SPECIFIER.

An alcGetString query of ALC_DEVICE_SPECIFIER or ALC_CAPTURE_DEVICE_SPECIFIER with a NULL device passed in will return a list of available devices. Each device name will be separated by a single NULL character and the list will be terminated with two NULL characters.

A request for ALC_DEFAULT_DEVICE_SPECIFIER on a system without an audio output device will result in NULL being returned.

A request for ALC_CAPTURE_DEFAULT_DEVICE_SPECIFIER on a system without an audio capture device will result in NULL being returned.

Table 6.3: String Query Tables

Name	***Description***
ALC_DEFAULT_DEVICE_SPECIFIER	The specifier string for the default device
ALC_DEVICE_SPECIFIER	The specifier string for the device
ALC_EXTENSIONS	A list of available context extensions separated by spaces.
ALC_CAPTURE_DEFAULT_DEVICE_SPECIFIER	The name of the default capture device
ALC_CAPTURE_DEVICE_SPECIFIER	The name of the specified capture device, or a list of all available capture devices if no capture device is specified.

In addition, printable error message strings are provided for all valid error tokens, including ALC_NO_ERROR, ALC_INVALID_DEVICE, ALC_INVALID_CONTEXT, ALC_INVALID_ENUM, ALC_INVALID_VALUE.

6.3.8. Integer Query

The application can query ALC for information using an integer query function.

```
void alcGetIntegerv(ALCdevice * deviceHandle, ALCenum token, ALCsizei size,
        ALCint *dest);
```

For some tokens, NULL is a legal deviceHandle. In other cases, specifying a NULL device will generate an ALC_INVALID_DEVICE error. The application has to specify the size of the destination buffer provided (in ALCint values). A NULL destination or a zero size parameter will cause ALC to ignore the query.

All tokens in table 6.1 (context creation attributes) and table 6.4 (context query types) are

valid for this call.

Table 6.4: Integer Query Types

Name	Description
ALC_ATTRIBUTES_SIZE	The size (number of ALCint values) required for a zero-terminated attributes list, for the current context. NULL is an invalid device.
ALC_ALL_ATTRIBUTES	Expects a destination of ALC_ATTRIBUTES_SIZE, and provides an attribute list for the current context of the specified device. NULL is an invalid device.
ALC_MAJOR_VERSION	The specification revision for this implementation (major version). NULL is an acceptable device.
ALC_MINOR_VERSION	The specification revision for this implementation (minor version). NULL is an acceptable device.
ALC_CAPTURE_SAMPLES	The number of capture samples available. NULL is an invalid device.

6.4. Shared Objects

For efficiency reasons, certain AL objects are shared across ALC contexts. At this time, AL buffers are the only shared objects.

6.4.1. Shared Buffers

Buffers are shared among contexts. The processing state of a buffer is determined by the dependencies imposed by all contexts, not just the current context. This includes suspended contexts as well as contexts that are processing.

6.4.2. Capture

Procedures and Functions

```
ALCdevice* alcCaptureOpenDevice(const ALCchar *deviceName, ALCuint freq,
                                ALCenum fmt, ALCsizei bufsize);

ALCboolean alcCaptureCloseDevice(ALCdevice *device);
void alcCaptureStart(ALCdevice *device);
void alcCaptureStop(ALCdevice *device);
void alcCaptureSamples(ALCdevice *device, ALCvoid *buf, ALCsizei samps);
```

Tokens

ALC_CAPTURE_DEFAULT_DEVICE_SPECIFIER
ALC_CAPTURE_DEVICE_SPECIFIER
ALC_CAPTURE_SAMPLES

Separate from traditional output devices, OpenAL may provide facilities for input, or "capture" of audio data from the user's environment. Capture devices run parallel to the rest of the library, and don't contain a context or possess most AL or ALC state. As such, they contain their own entry points that represent a greatly simplified interface. Capture is handled as a polling, non-blocking sub-API within OpenAL.

The alcCaptureOpenDevice function allows the application to connect to a capture device. To obtain a list of all available capture devices, use alcGetString to retrieve ALC_CAPTURE_DEVICE_SPECIFIER with a NULL device specified – a list of all capture devices will be returned (each name will be NULL-terminated, and the list will be terminated with two NULL characters). Retrieving ALC_CAPTURE_DEVICE_SPECIFIER with a valid capture device specified will result in the name of that device being returned as a single NULL-terminated string.

```
ALCdevice* alcCaptureOpenDevice(const ALCchar *deviceName,
                    ALCuint freq, ALCenum fmt, ALCsizei bufsize);
```

If the function returns NULL, then no sound driver/device has been found, or the requested format could not be fulfilled.

The "deviceName" argument is a null terminated string that requests a certain device or device configuration. If NULL is specified, the implementation will provide an implementation specific default. The "freq" and "fmt" arguments specify the format that audio data will be presented to the application, and match the values that can be passed to alBufferData. The implementation is expected to convert and resample to this format on behalf of the application. The "bufsize" argument specifies the number of sample frames to buffer. For example, requesting a format of AL_FORMAT_STEREO16 and a buffer size of 1024 would require the AL to store up to 1024 * 4 bytes of audio data. Note that the implementation may use a larger buffer than requested if it needs to, but the implementation will set up a buffer of at least the requested size.

Specifying a compressed or extension-supplied format may result in failure, even if the extension is supplied for rendering.

The alcCaptureCloseDevice function allows the application to disconnect from a capture device.

```
ALCboolean alcCaptureCloseDevice(ALCdevice *deviceHandle);
```

The return code will be ALC_TRUE or ALC_FALSE, indicating success or failure. If deviceHandle is NULL or invalid, an ALC_INVALID_DEVICE error will be generated. Once closed, a deviceHandle is invalid.

Audio capture

Once a capture device has been opened via alcCaptureOpenDevice, it is made to start recording audio via the alcCaptureStart entry point:

void alcCaptureStart(ALCdevice *device);

Once started, the device will record audio to an internal ring buffer, the size of which was specified when opening the device.

The application may query the capture device to discover how much data is currently available via the alcGetInteger with the ALC_CAPTURE_SAMPLES token. This will report the number of sample frames currently available.

When the application feels there are enough samples available to process, it can obtain them via the alcCaptureSamples entry point:

void alcCaptureSamples(ALCdevice *device, ALCvoid *buf, ALCsizei samps);

The "buf" argument specifies an application-allocated buffer that must contain at least "samps" sample frames. Exactly “samps” number of samples will be retrieved by this call and placed in the buffer, without blocking. If fewer than “samps” frames are available, then an ALC_INVALID_VALUE error is set. If more than “samps” frames are available, the extra samples are retained by OpenAL and are available to be retrieved at a later time. The implementation may defer conversion and resampling until this point. Requesting more sample frames than are currently available is an error.

If the application doesn't need to capture more audio for an amount of time, they can halt the device without closing it via the alcCaptureStop entry point:

void alcCaptureStop(ALCdevice *device);

The implementation is encouraged to optimize for this case. The amount of audio samples available after restarting a stopped capture device is reset to zero. The application does not need to stop the capture device to read from it.

7. Appendix: Extensions

Extensions are a way to provide for future expansion of the OpenAL API. Typically, extensions are specified and proposed by a vendor, and can be treated as vendor neutral if no intellectual property restrictions apply. Extensions can also be specified as, or promoted to be, ARB extensions, which is usually the final step before adding a tried and true extension to the core API. ARB extensions, once specified, have mandatory presence for backwards compatibility.

7.1. Extension Query

To use an extension, the application will have to obtain function addresses and enumeration values. Before an extension can be used, the application should verify the presence of an extension using alIsExtensionPresent. The application can then retrieve the address (function pointer) of an extension entry point using alGetProcAddress. Extensions and entry points can be Context-specific, and the application can not count on an extension being available unless alIsExtensionPresent returns AL_TRUE. The application also has to maintain pointers on a per-context basis.

ALboolean alIsExtensionPresent(const ALchar *extName);

Returns AL_TRUE if the given extension is supported for the current context, AL_FALSE otherwise. extName is not case sensitive – the implementation will convert the name to all upper-case internally (and will express extension names in upper-case).

7.2. Retrieving Function Entry Addresses

void *alGetProcAddress(const ALchar *funcName);

Returns NULL if no entry point with the name funcName can be found. Implementations are free to return NULL if an entry point is present, but not applicable for the current context. However the specification does not guarantee this behavior.

Applications can use alGetProcAddress to obtain core API entry points, not just extensions. This is the recommended way to dynamically load and unload OpenAL DLL's as sound drivers.

7.3. Retrieving Enumeration Values

To obtain enumeration values for extensions, the application has to use alGetEnumValue of an extension token. Enumeration values are defined within the OpenAL name space and allocated according to specification of the core API and the extensions, thus they are context-independent.

ALuint alGetEnumValue(const ALchar *enumName);

Returns 0 if the enumeration can not be found, and sets an AL_INVALID_VALUE error condition. The presence of an enum value does not guarantee the applicability of an extension to the current context. A non-zero return indicates merely that the implementation is aware of the existence of this extension.

8. Appendix: Extension Process

There are two ways to suggest an Extension to AL or ALC. The simplest way is to write an ASCII text that matches the following template:

RFC: rfc-iiyymmdd-nn
Name: (indicating the purpose/feature)
Maintainer: (name and spam-secured e-mail)
Date: (last revision)
Revision: (last revision)

new enums
new functions

description of operation

Such an RFC can be submitted on the OpenAL discussion list (please use RFC in the Subject line), or send to the maintainer of the OpenAL specification. If you are shipping an actual implementation as a patch or as part of the OpenAL CVS a formal writeup is recommend. In this case, the Extension has to be described as part of the specification, which is maintained in OpenOffice.org format.

9. Appendix: 1.0-Compatible Extensions

This section shows how to access most of the expanded functionality of OpenAL 1.1 from an OpenAL 1.0 application. It is not necessary to query for these extensions from an OpenAL 1.1 application (use the ALC_MAJOR_VERSION and ALC_MINOR_VERSION properties to verify the implementation version).

9.1. ALC_EXT_CAPTURE

An OpenAL 1.1 implementation will always support the ALC_EXT_CAPTURE extension. This allows an application written to the OpenAL 1.0 specification to access the capture abilities expressed in section 6.4.2.

Capture Procedures and Functions

```
ALCdevice* alcCaptureOpenDevice(const ALCchar *deviceName, ALCuint freq,
                    ALCenum fmt, ALCsizei bufsize);

ALCboolean alcCaptureCloseDevice(ALCdevice *device);
void alcCaptureStart(ALCdevice *device);
void alcCaptureStop(ALCdevice *device);
void alcCaptureSamples(ALCdevice *device, ALCvoid *buf, ALCsizei samps);
```

Capture Tokens

```
ALC_CAPTURE_DEFAULT_DEVICE_SPECIFIER
ALC_CAPTURE_DEVICE_SPECIFIER
ALC_CAPTURE_SAMPLES
```

9.2. AL_EXT_OFFSET

An OpenAL 1.1 implementation will always support the AL_EXT_OFFSET extension. This allows an application written to the OpenAL 1.0 specification to access the offset abilities expressed in section 4.3.2.

Offset tokens:

```
AL_SEC_OFFSET
AL_SAMPLE_OFFSET
AL_BYTE_OFFSET
```

9.3. AL_EXT_LINEAR_DISTANCE

An OpenAL 1.1 implementation will always support the

AL_EXT_LINEAR_DISTANCE extension. This allows an application written to the OpenAL 1.0 specification to access the offset abilities expressed in sections 3.4.3 and 3.4.4.

Tokens:

AL_LINEAR_DISTANCE
AL_LINEAR_DISTANCE_CLAMPED

9.4. AL_EXT_EXPONENT_DISTANCE

An OpenAL 1.1 implementation will always support the AL_EXT_EXPONENT_DISTANCE extension. This allows an application written to the OpenAL 1.0 specification to access the offset abilities expressed in sections 3.4.5 and 3.4.6.

Tokens:

AL_EXPONENT_DISTANCE
AL_EXPONENT_DISTANCE_CLAMPED

9.5. ALC_ENUMERATION_EXT

An OpenAL 1.1 implementation will always support the ALC_ENUMERATION_EXT extension. This extension provides for enumeration of the available OpenAL devices through alcGetString. An alcGetString query of ALC_DEVICE_SPECIFIER with a NULL device passed in will return a list of devices. Each device name will be separated by a single NULL character and the list will be terminated with two NULL characters.