OpenAL Specification and Reference

OpenAL Specification and Reference
Version 1.0 Draft Edition
Published June 2000

Table of Contents

List of Examples

Chapter 1. Introduction

1.1. Formatting and Conventions

This API Specification and Reference uses a style that is a blend of the OpenGL v1.2 specification and the OpenGL Programming Guide, 2nd ed. Conventions: 'T' is used to designate a type for those functions which exist in multiple signatures for different types. 'Object' is used to designate a target Object for those functions which exist in multiple versions for different Object categories. The 'al' and 'AL_' prefix is omitted throughout the document.

> **Annotation (Terminology):** "State" refers to state within the context of the OpenAL state machine description of the OpenAL implementation. "Objects" refer to OpenAL primitives. "Attribute" refers to attributes of OpenAL Objects. Attributes of a OpenAL Objects are one part of the OpenAL state. Some attributes are not specific to single objects, but apply to the entire context. "Parameter" is used for function arguments that might or might not be attributes, in particular for command arguments that are not stored as state. The use of "Property" is to be avoided.

Revision History

Revision 1.8 / 1.7. / 1.6 / 1.5	September-August 2000	Revised by: bk
Final Draft for Public Review		
Revision 1.4	June 2000	Revised by: bk
First Draft for Public Review		
Revision 1.2	March 2000	Revised by: mkv
Draft released for GDC		

1.2. 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, 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.

1.3. 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 (e.g. 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.

> **Annotation (OpenAL and OpenGL use):** Often, OpenAL will be used to render a 3D audio environment matched by a 3D visual scenery. For this purpose, OpenAL is meant to be a seamlessly integrating complement to OpenGL. OpenAL state can be updated in sync with the OpenGL or video updates (synchronized), or in timesteps independent of the graphics framerate. Audio rendering loops usually update the current locations of the sources and the listener, updates global settings, and manages buffers.

1.4. 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 degress. The OpenAL implementors 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 eludicate how it changes, and to indicate what its effects are.

> **Annotation (Native audio APIs):** Implementors can choose to implement OpenAL on top of an existing native audio API.

1.5. 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 conformant 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.

1.6. 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 will be an OpenAL set of conformance tests available along with the open source sample implementation. Vendors and individuals are encouraged to specify and implement extensions to OpenAL in the same way OpenGL is extensible. 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 openal.org[1]. OpenAL is also available from the the OpenAL CVS repository[2]. For more information on how to get OpenAL from CVS also see Loki Software CVS[3].

1.7. Architecture Review and Acknowledgements

Like OpenGL, OpenAL is meant to evolve through a joined effort of implementators and application programmers meeting in regular sessions of an Architecture Review Board (ARB). As of this time the ARB has not yet been set up. Currently, the two companies committed to implementing OpenAL drivers have appointed two contacts responsible for preparing the specification draft.

Consequently OpenAL is a cooperative effort, one in a sequence of earlier attempts to create a cross-platform audio API. The current authors/editors have assembled this draft of the specification, but many have, directly and indirectly, contributed to the content of the actual document. The following list (in all likelihood incomplete) gives in alphabetical order participants in the discussion and contributors to the specification processs and related efforts: Juan Carlos Arevalo Baeza, Jonathan Blow, Keith Charley, Scott Draeker, John Grantham, Jacob Hawley, Garin Hiebert, Carlos Hasan, Nathan Hill, Bill Huey, Mike Jarosch, Jean-Marc Jot, Maxim Kizub, John Kraft, Bernd Kreimeier, Ian Ollmann, Rick Overman, Sean L. Palmer, Pierre Phaneuf, Terry Sikes, Joseph Valenzuela, Michael Vance, Carlo Vogelsang

Notes

1. http://wwww.openal.org/
2. http://cvs.lokigames.com/cgi-bin/cvsweb.cgi/openal/
3. http://cvs.lokigames.com/

Chapter 2. OpenAL Operation

2.1. OpenAL Fundamentals

OpenAL (henceforth, the "AL") is concerned only with rendering audio into an output buffer, and primarily meant for spatialized audio. There is no support for reading audio input from buffers at this time, and no support for MIDI and other components usually associated with audio hardware. Programmers must relay on other mechanisms to obtain audio (e.g. voice) input or generate music.

The AL 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 AL 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. There are normally far more sources defined than buffers. 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 un-compressed 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. The influence of all the sources from the perspective of the listener is mixed and played for the user.

2.1.1. Primitive Types

As AL is meant to allow for seamless integration with OpenGL code if needed, the AL primitive (scalar) data types mimic the OpenGL data types. Guaranteed minimum sizes are stated for OpenGL data types (see table 2.2 of the OpenGL 1.2 Specification), but the actual choice of C datatype is left to the implementation. All implementations on a given binary architecture, however, must use a common definition of these datatypes.

Note that this table uses explicit AL prefixes for clarity, while they might be omitted from the rest of the document for brevity. GCC equivalents are given for IA32, i.e. a portable and widely available compiler on the most common target architecture.

Table 2-1. AL Primitive Data Types

AL Type	Description	GL Type	GCC IA32
ALboolean	8-bit boolean	GLboolean	unsigned char
ALbyte	signed 8-bit 2’s-complement integer	GLbyte	signed char
ALubyte	unsigned 8-bit integer	GLubyte	unsigned char

AL Type	Description	GL Type	GCC IA32
ALshort	signed 16-bit 2's-complement integer	GLshort	short
ALushort	unsigned 16-bit integer	GLushort	unsigned short
ALint	signed 32-bit 2's-complement integer	GLint	int
ALuint	unsigned 32-bit integer	GLuint	unsigned int
ALsizei	non-negative 32-bit binary integer size	GLsizei	int
ALenum	enumerated 32-bit value	GLenum	unsigned int
ALbitfield	32 bit bitfield	GLbitfield	unsigned int
ALfloat	32-bit IEEE754 floating-point	GLfloat	float
ALclampf	Same as ALfloat, but in range [0, 1]	GLclampf	float
ALdouble	64-bit IEEE754 floating-point	GLdouble	double
ALclampd	Same as ALdouble, but in range [0, 1]	GLclampd	double

Annotation on Type Sizes: It would be desirable to guarantee the bit size of AL data types, but this might affect the mapping to OpenGL types for which the OpenGL specification only guarantees a minimum size.

Annotation on 64bit integral: It would be desirable to define ulong and long, but again we defer to OpenGL in this decision.

Annotation on Enumeration: enum is not a C or C++ enumeration, but implemented as C preprocesor defines. This makes it easier to handle extensions to the AL namespace, in particular in dealing with delays in distributing updated reference headers.

2.1.2. Floating-Point Computation

Any representable floating-point value is legal as input to a AL 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 AL interruption or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing an 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 GL interruption or termination.

2.2. AL State

The AL maintains considerable state. This documents 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 the AL performs on the implied output buffer, the outbut buffer is not part of the AL state. Certain states of AL objects (e.g. buffer states with respect to queueing) are introduced for discussion purposes, but not exposed through the API.

2.3. AL Command Syntax

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

> **Annotation (Not all types supported yet):** At this time AL does not support the full flexibility that OpenGL offers. Certain entry points are supported only for some data types. In general, AL tends to use less entry points, using setter commands that use the same tokens as the matching query commands.

2.4. Basic AL Operation

AL 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 / AL program, most of the audio programming will be in two places in the code: initialization of the program, and the rendering loop. An OpenGL / AL 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 AL.

Example 2-1. Initialization Example

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

2.5. AL Errors

The AL 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

```
enum  GetError (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 GetError 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 GetError returns NO_ERROR then there has been no detectable error since the last call to GetError (or since the AL was initialized).

> **Annotation (Only First Error):** Like OpenGL AL will ignore subsequent errors once an error conditation has been encountered.

Error codes can be mapped to strings. The GetString 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.

> **Annotation/ Verbose Error String:** There is no need to maintain a separate GetErrorString function (inspired by the proposed gluGetErrorStrings) as the existing GetString entry point can be used.

Table 2-2. Error Conditions

Name	Description
NO_ERROR	"No Error" token.
INVALID_NAME	Invalid Name parameter.
INVALID_ENUM	Invalid parameter.
INVALID_VALUE	Invalid enum parameter value.
INVALID_OPERATION	Illegal call.
OUT_OF_MEMORY	Unable to allocate memory.

The table summarizes the AL errors. Currently, when an error flag is set, results of AL operations are undefined only if OUT_OF_MEMORY has occured. 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 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 AL objects such as Buffers and Sources are checked for validity. If an invalid name parameter is specified in an AL command, an INVALID_NAME error will be generated, and the command is ignored.

If a negative integer is provided where an argument of type sizei is specified, the error INVALID_VALUE results. The same error will result from attempts to set integral and floating point values for attributes exceeding the legal range for these. The specification does not guarantee that the implementation emits 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. AL responds to any such illegal command with an 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 OUT_OF_MEMORY may be generated. This can also happen independent of recent commands if AL 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 explicitely described in this specification.

2.6. 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  Enable ( enum  target );
```

```
void  Disable ( enum  target );
```

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

```
boolean  IsEnabled ( enum  target );
```

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

At this time, this mechanism is not used. There are no valid targets.

> **Annotation (Enable/Disable):** Currently, AL is controlled exploiting existing commands. For example, to disable sound output but not processing, the Listener can be muted setting GAIN to zero. Selecting NONE as the distance model disables distance attenuation. Setting DOPPLER_FACTOR to zero disables the Doppler Effect. A redundant mechanism to accomplish the same is not needed.

2.7. Object Paradigm

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

2.7.1. Object Categories

AL has three primary categories of Objects:

- one unique Listener per Context
- multiple Buffers shared among Contexts
- multiple Sources, each local to a Context

In the following, "{Object}" will stand for either Source, Listener, or Buffer.

2.7.2. Static vs. Dynamic Objects

The vast majority of AL objects are dynamic, and will be created on application demand. There are also AL 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 AL.

2.7.3. 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 (uint). 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 AL objects, but no guarantees are made across different classes of objects. Objects like the Listener that are unique (singletons) do not require, and do not have, an integer "name".

2.7.4. Requesting Object Names

AL provides calls to obtain Object Names. The application requests a number of Objects of a given category using Gen{Object}s. If the number n of Objects requested is negative, an INVALID_VALUE error will caused. The actual values of the Names returned are implementation dependent. No guarantees on range or value are made. Unlike OpenGL OpenAL does not offer alternative means to define (bind) a Name.

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  Gen{Object}s ( sizei  n ,  uint *  objectNames );
```

Requesting zero names is a legal NOP. Requesting a negative number of names causes an INVALID_VALUE error. AL will respond with an OUT_OF_MEMORY if the application requests too many objects. The specification does not guarantee that the AL implementation will allocate all resources needed for the actual objects at the time the names are reserved. In many cases (Buffers) this could only be implemented by worst case estimation. Allocation of names does not guarantee that all the named objects can actually be used.

Annotation (No application selected Names): Unlike GL, applications are not free to choose Names, all Names have to be requested. Aside from possible benefits for the implementation, and avoidance of errors in projects that have many modules using the AL implementation (a problem encountered in GL, when the two generation mechanisms are mixed), this also leaves open the door to feed different kinds of objects by Name through the same API entry points.

Annotate (Negative/zero sizei): The specification does not guarantee that sizei is an unsigned integer, but legal values have to be non-negative. However, requesting zero names is a legal NOP.

2.7.5. Releasing Object Names

AL provides calls to the application to release Object Names using Delete{Object}s, implicitly requesting deletion of the Objects associated with the Names released. If the number n of Objects named is negative, an INVALID_VALUE error will be caused. If one or more of the specified Names is not valid, an INVALID_NAME error will be caused. Implementation behavior following any error is undefined.

Once deleted (even if an error occured on deletion), the Names are no longer valid for use with any AL function calls including calls to Delete{Objects}s. Any such use will cause an INVALID_NAME error.

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

```
void Delete{Object}s(sizei n, uint *objectNames);
```

Annotation: GenXXX and DeleteXXX can not reasonably be expected to be used for controlling driver-side resource management from the application. A driver might never release a Source once allocated during the lifetime of the application.

2.7.6. Validating an Object Name

AL provides calls to validate the Name of an Object. The application can verify whether an Object Name is valid using the Is{Object} query. There is no vector (array) version of this function as it defeats the purpose of unambiguous (in)valdiation. Returns TRUE if id is a valid Object Name, and FALSE otherwise. Object Names are valid between request (Gen{Object}s) and release (Delete{Object}s). Is{Object} does not distinguish between invalid and deleted Names.

```
boolean Is{Object}(uint objectName);
```

2.7.7. Setting Object Attributes

For AL Objects, calls to control their attributes are provided. These depend on the actual properties of a given Object Category. The precise API is discussed for each

category, below. Each AL command affecting the state of a named Object is usually of the form

```
void {Object}{n}{sifd}{v} ( uint objectName , enum paramName , T values );
```

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

```
void {Object}{n}{sifd}{v} ( enum paramName , T values );
```

For example, the Listener3d command would not require an (integer) objectName argument.

The objectName specifies the AL object affected by this call. Use of an invalid Name will cause an INVALID_NAME error.

The Object's Attribute to be affected has to be named as paramName. AL parameters applicable to one category of Objects are not necessarily legal for another catetgory of AL Objects. Specification of a parameter illegal for a given object will cause an 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 INVALID_VALUE error.

Any command that causes an error is a NOP.

2.7.8. Querying Object Attributes

For named and for unique AL Objects, calls to query their current attributes are provided. 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 complementing attribute setting function.

```
void Get{Object}{n}{sifd}{v} ( uint objectName , enum paramName , T * destination );
```

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

```
void Get{Object}{n}{sifd}{v} ( enum paramName , T * destination );
```

The precise API is discussed for each category separately, below. Unlike their matching mutators, Query functions for non-scalar properties (vectors etc.) are only available in array form.

Use of an invalid Name will cause an INVALID_NAME error. Specification of an illegal parameter type (token) will cause an INVALID_ENUM error. A call with a destination NULL pointer will be quietly ignored. The AL state will not be affected by errors. In case of errors, destination memory will not be changed.

2.7.9. Object Attributes

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

The general form in which this document describes parameters is

Table 2-3. {Object} Parameters

Name	Signature	Values	Default
paramName	T	range or set	scalar or n-tupel

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

Chapter 3. State and State Requests

The majority of AL state is associated with individual AL objects, and has to be set and queried referencing the objects. However, some state - e.g. processing errors - is defined context specific. AL has global state that affects all objects and processing equally. This state is set using a variety of functions, and can be queried using query functions. The majority of queries has to use the interface described in "Simple Queries".

3.1. Querying AL State

3.1.1. Simple Queries

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

```
void  GetBooleanv ( enum  paramName ,  boolean *  dest );

void  GetIntegerv ( enum  paramName ,  int *  dest );

void  GetFloatv ( enum  paramName ,  float *  dest );

void  GetDoublev ( enum  paramName ,  double *  dest );
```

Legal values are e.g. DOPPLER_FACTOR, DOPPLER_VELOCITY, DISTANCE_MODEL.

NULL destinations are quietly ignored. INVALID_ENUM is the response to errors in specifying paramName. The amount of memory required in the destination depends on the actual state requested. Usually, state variables are returned in only one or some of the formats above.

To query state controlled by Enable / Disable there is an additional IsEnabled function defined (see "Controlling AL Execution").

3.1.2. Data Conversions

If a Get command is issued that returns value types different from the type of the value being obtained, a type converswion is performed. If GetBooleanv is called, a floating-point or integer value converts to FALSE if and only if it is zero (otherwise it converts to TRUE). If GetIntegerv is called, a boolean value is interpreted as either 1 or 0, and a floating-point value is rounded to the nearest integer. If GetFloatv is called, a boolean value is interpreted as either 1.0 or 0.0, an integer is coerced to floating point, and a double-presicion foating-point value is converted to single precision. Analogous conversions are carried out in the case of GetDoublev. If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value is representable using the requested type is returned.

> **Annotation (Query of Modes):** Modes (e.g. the current distance model) can be queried using the respective tokens. The recommended query command is GetIntegerv.

3.1.3. String Queries

The application can retrieve state information global to the current AL Context. GetString will return a pointer to a constant string. Valid values for param are VERSION, RENDERER, VENDOR, and EXTENSIONS, as well as the error codes defined by AL. The application can use GetString to retrieve a string for an error code.

```
const ubyte *  GetString ( enum  paramName);
```

3.2. Time and Frequency

By default, AL 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

AL does not define the units of measurement for distances. The application is free to use meters, inches, or parsecs. AL provides means for simulating the natural attenuation of sound according to distance, and to exagerate 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. AL 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.

Annotation (No DistanceFactor): AL does not provide a global, per Context DISTANCE_FACTOR, as all calculations are scale invariant.

Annotation (Distance Calculations): The specification does not enforce that distances are calculated using the euclidean norm, to permit using computationally cheaper approximations. Implementations that opt to use approximations might cause audible artifacts. The specification does not enforce any upper limits on distance calculation errors yet.

Annotation (DS3D DistanceFactor): The DS3D documentation explicitely states that the default unit is 1 meter. AL does not specify any units. AL calculations are scale invariant. The main purpose of the DS3D DistanceFactor (modifying Doppler Shift by scaling velocity but not reference velocity) is accomplished using DOPPLER_VELOCITY in AL. AL does not have an equivalent to A3D's SetUnitsPerMeter either.

3.4. Attenuation By Distance

Samples usually use the entire dynamic range of the chosen format / encoding, independent of their real world intensity. In other words, 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.

AL currently supports three modes of operation with respect to distance attenuation. It supports two distance-dependent attenuation models, one which is similar to the IASIG I3DL2 (and DS3D) model. The application choses one of these two models (or can chose to disable distance-dependent attenuation effects model) on a per-context basis.

```
void  DistanceModel ( enum  modelName );
```

Legal arguments are NONE, INVERSE_DISTANCE, and INVERSE_DISTANCE_CLAMPED. NONE bypasses all distance attenuation calculation for all Sources. The implementation is expected to optimize this situation. INVERSE_DISTANCE_CLAMPED is the DS3D model, with REFERENCE_DISTANCE indicating both the reference distance and the distance below which gain will be clamped. INVERSE_DISTANCE is equivalent to the DS3D model with the exception that REFERENCE_DISTANCE does not imply any clamping. The AL implementation is still free to apply any range clamping as necessary. The current distance model chosen can be queried using GetIntegerv and DISTANCE_MODEL.

Annotation (Inverse Square Law): The "inverse square law" used in physics applies to sound intensity (energy), which is proportional to the square of linear gain (amplitude). Thus the inverse distance model describes a physically correct inverse square behavior if ROLLOFF_FACTOR is set to 1.0.

Annotation (Enable/Disable Attenuation): As ROLLOFF_FACTOR is a per-Source attribute, setting it to zero can not be used to globally enable or disable distance attenuation, which (e.g. when using tables) can be resource intensive. Using Enable/Disable/IsEnabled with a DISTANCE_ATTENUATION token is redundant with respect to the possibility that support for different distance models might be desired at a later time.

3.4.1. Inverse Distance Rolloff Model

The following formula describes the distance attenutation defined by the Rolloff Attenutation Model, as logarithmic calculation.

```
    G_dB = GAIN - 20*log10(1 + ROLLOFF_FACTOR*(dist-
REFERENCE_DISTANCE) / REFERENCE_DISTANCE );
    G_dB = min(G_dB,MAX_GAIN);
    G_dB = max(G_dB,MIN_GAIN);
```

The REFERENCE_DISTANCE parameter used here is a per-Source attribute that can be set and queried using the REFERENCE_DISTANCE token. REFERENCE_DISTANCE is the distance at which the Listener will experience GAIN (unless the implementation had to clamp effective GAIN to the available dynamic range). 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. The implementation is free to optimize for a ROLLOFF_FACTOR value of 0, which indicates that the application does not wish any distance attenuation on the respective Source.

Annotation (Linear Calculation): The logarithmic formula above is equivalent to

```
G  = gain_linear / ( 1 + ROLLOFF_FACTOR*((dist-
REFERENCE_DISTANCE)/REFERENCE_DISTANCE) );
G = min(G,max_gain_linear);
G = max(G,min_gain_linear);
```

with linear gains calculated from the logarithmic GAIN, MIN_GAIN, MAX_GAIN accordingly. By means of explanation: linear GAIN is applied to the sample, which describes an amplitude ultimately (DAC) converted into voltage. The actual power of the signal is proportional to the square of the amplitude (voltage). Logarithmic measurement is done by comparing the actual power with a reference value, i.e. the power (e.g in Watt) at the reference distance. The original Bel unit of measure (named after Alexander Graham Bell) was defined to account for the logarithmic response of the human ear: our subjective impression of "loudness" is not linear in the power of the acoustic signal. For practical purposes (range of volumes the human ear can handle) the deciBel (dB) is a better unit:

dB = 10 * log(P/P0) = 10 * log(sqr(A/A0) = 20 * log(A/A0)

Common power/amplitude ratios and attenuations per distance are:

Table 3-1. Logarithmic Scale and Gain

Distance	Attenuation	Power Ratio	Amplitude Ratio
REF	0dB	1:1	1:1
2*REF	-6dB	1:4	1:2
4*REF	-12dB	1:16	1:4
8*REF	-18dB	1:64	1:8
0.5*REF	6dB	2:1	4:1
0.25*REF	12dB	4:1	16:1

The logarithmic gain will drop from zero (linear gain 1) to negative infinity (approaching linear gain 0). A linear gain of zero can not be represented logarithmically. Any doubling of the reference distance will add another -6dB (i.e. 6dB of attenuation). This approximates an inverse square law falloff of signal power with distance, as long as the distance exceeds the reference distance.

Annotation (Rolloff quantization): Implementations that use lookup tables to speed up distance attenuation calculation may opt to map ROLLOFF_FACTOR to a limited set of internally used values, to minimize expense of per-Source calculations and setup/memory costs.

Annotation (Gain Clamping): In the absence of user MIN_GAIN and MAX_GAIN selections, clamping is implied by implementation constraints, and clamping behavior might change. The AL implementation should not clamp intermediate values of effective gain to unit range. Any clamping, if necessary, should be applied at the latest possible stage. In other words, GAIN>1 is perfectly valid as the implementation is free to clamp the value as needed for maximum mixing accuracy and to account for the actual dynamic range of the output device.

Annotation (Extended Dynamic Range): For applications that change GAIN but do not want to adjust ROLLOFF_FACTOR and REFERENCE_DISTANCE to account for different ranges, the separation in this distance model might allow for more intuitive adjustments: If we put a damper on the jet engine by lowering GAIN, we still want the listener to perceive the full volume, but now at a closer distance, without changing the reference distance.

3.4.2. Inverse Distance Clamped Model

This is essentially the Inverse Distance model, extended to guarantee that for distances below REFERENCE_DISTANCE, gain is clamped. This mode is equivalent to the IASIG I3DL2 (and DS3D) distance model.

```
dist = max(dist,REFERENCE_DISTANCE);
dist = min(dist,MAX_DISTANCE);
G_dB = GAIN - 20*log10(1 + ROLLOFF_FACTOR*(dist-
REFERENCE_DISTANCE)/REFERENCE_DISTANCE)
G_dB = min(G_dB,MAX_GAIN);
G_dB = max(G_dB,MIN_GAIN);
```

Annotation (DS3D MIN_DISTANCE): The DS3D attenuation model is extended by an explicit clamping mechanism. REFERENCE_DISTANCE is equivalent to DS3D MIN_DISTANCE if the INVERSE_DISTANCE_CLAMPED mode is used.

Annotation (High Frequency Rolloff): To simulate different atmospheric conditions, a frequency dependent attenuation is used in A3D and EAX. At this time AL does not have a mechanism to specify lowpass filtering parameterized by distance.

3.5. Evaluation of Gain/Attenuation Related State

While amplification/attenuation commute (mulplication of scaling factors), clamping operations do not. The order in which various gain related operations are applied is: Distance attenuation is calculated first, including minimum (REFERENCE_DISTANCE) and maximum (MAX_DISTANCE) thresholds. If the Source is directional (CONE_INNER_ANGLE less than CONE_OUTER_ANGLE), an angle-dependent attenuation is calculated depending on 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 GAIN. The effective GAIN computed this way is compared against MIN_GAIN and MAX_GAIN thresholds. The result is guaranteed to be clamped to

[MIN_GAIN, 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.

3.6. No Culling By Distance

With the DS3D compatible Inverse Clamped Distance Model, AL provides a per-Source 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 proccessing, based on distance. However, AL does not support culling a Source from processing based on a distance threshold.

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

Annotation (No MAX_DISTANCE plus MUTE): AL does support the AUDIBLE mode with MAX_DISTANCE (clamping), but does not support culling. Applications that employ this DS3D feature will have to perform their own distance calculation and mute the source (Source GAIN equals zero) as desired.

3.7. 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 application can increase or decrease that frequency shift by specifying the scaling factor AL should apply to the result of the calculation.

The Doppler Effect as implemented by AL is described by the formula below. Effects of the medium (air, water) moving with respect to listener and source are ignored. DOPPLER_VELOCITY is the propagation speed relative to which the Source velocities are interpreted.

VD: DOPPLER_VELOCITY
DF: DOPPLER_FACTOR
vl: Listener velocity (scalar, projected on source-listener vector)
vs: Source verlocity (scalar, projected on source-listener vector)
f: Frequency in sample
f': effective Doppler shifted frequency

$$f' = DF * f * (VD - vl) / (VD + vs)$$

vl<0, vs>0 : source and listener approaching each other
vl>0, vs<0 : source and listener moving away from each other

The implementation has to clamp the projected Listener velocity vl, if abs(vl) is greater or equal VD. It similarly has to clamp the projected Source velocity vs if abs(vs) is greater or equal VD.

There are two API calls global to the current context that provide control of the two related parameters. DOPPLER_FACTOR is a simple scaling to exaggerate or deemphasize the Doppler (pitch) shift resulting from the calculation.

```
void  DopplerFactor ( float  dopplerFactor );
```

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

> **Annotation (No Enable):** There is currently no mechanism to switch on/off Doppler calculation using e.g. a DOPPLER_SHIFT token and Enable/Disable. For the time being, DopplerFactor(0) may be used to signal to the implementation that no Doppler Effect calculation is required.

DOPPLER_VELOCITY allows the application to change the reference (propagation) velocity used in the Doppler Effect calculation. This permits the application to use a velocity scale appropriate to its purposes.

```
void  DopplerVelocity ( float  dopplerVelocity);
```

A negative or zero value will result in an INVALID_VALUE error, the command is then ignored. The default value is 1. The current setting can be queried using GetFloatv and DOPPLER_VELOCITY.

> **Annotation (No Sideeffects on Delay):** To permit independent control of Doppler Effect as opposed to other, sound wave propagation related effects (delays, echos, reverbration), DOPPLER_VELOCITY is not taken into account for any other calculation than Doppler Shift.

> **Annotation (SetUnitsPerMeter):** DOPPLER_VELOCITY accomplishes the purposes of DS3D scaling parameters in a straightforward way, without introducing the undesirable connotations of real world units.

Chapter 4. Listener and Sources

4.1. Basic Listener and Source Attributes

This section introduces basic attributes which can be set both for the Listener object and for Source objects.

The AL Listener and Sources have attributes to describe their position, velocity and orientation in three dimensional space. AL 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.

Table 4-1. Listener/Source Position

Name	Signature	Values	Default
POSITION	3fv, 3f	any except NaN	{ 0.0f, 0.0f, 0.0f }

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

> **Annotation (No Transformation):** AL does not support transformation operations on Objects. Support for transformation matrices is not planned.

Table 4-2. Listener/Source Velocity

Name	Signature	Values	Default
VELOCITY	3fv, 3f	any except NaN	{ 0.0f, 0.0f, 0.0f }

Description: 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 VELOCITY does not affect its position. AL 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.

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.

Table 4-3. Listener/Source Gain (logarithmic)

Name	Signature	Values	Default
GAIN	f	0.0f, (0.0f, any	1.0f

Description: 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 un-attenuated. A GAIN value of 0.5 is equivalent to an attenuation of 6 dB. The value zero equals silence (no output). 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.

GAIN larger than 1 (amplification) is permitted for Source and Listener. However, the implementation is free to clamp the total gain (effective gain per source times listener gain) to 1 to prevent overflow.

Annotation/ Effective Minimal Distance: Presuming that the sample uses the entire dynamic range of the encoding format, an effective gain of 1 represents the maximum volume at which a source can reasonably be played. During processing, the implementation combines the Source GAIN (or MIN_GAIN, if set and larger) with distance based attenuation. The distance at which the effective gain is 1 is equivalent to the DirectSound3D MIN_DISTANCE parameter. Once the effective gain has reached the maximum possible value, it will not increase with decreasing distance anymore.

Annotation (Muting a Context): To mute the current context, simply set Listener GAIN to zero. The implementation is expected to optimize for this case, calculating necessary (offset) updates but bypassing the mixing and releasing hardware resources. The specification does not guarantee that the implementation will release hardware resources used by a muted context.

Annotation (Muting a Source): To mute a Source, set Source GAIN to zero. The AL implementation is encouraged to optimize for this case.

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 AL 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 / pickup point and orientation, and other parameters that affect the output stream.

It is entirely up to the driver and hardware configuration, i.e. the installation of AL as part of the operating system and hardware setup, whether the output stream is generated for headphones or 2 speakers, 4.1 speakers, or other arrangements, whether (and which) HRTF's are applied, etc..

Annotation (Listener Anatomy): The API is ignorant with respect to the real world listener, it does not need to make assumptions on the listening capabilities of the user, its species or its number of ears. It only describes a scene and the position of the listener in this scene. It is the AL implementation that is designed for humans with ears on either side of the head.

Annotation (Listener State Evaluation): Some Listener state (GAIN) affects only the very last stage of sound synthesis, and is thus applied to the sound stream as sampled at the Listener position. Other Listener state is applied earlier. One example is Listener velocity as used to compute the amount of Doppler pitch-shifting applied to each source:

In a typical implementation, pitch-shifting (sample-rate conversion) might be the first stage of the audio processing for each source.

4.2.1. Listener Attributes

Several Source attributes also apply to Listener: e.g. POSITION, VELOCITY, GAIN. In addition, some attributes are listener specific.

Table 4-4. Listener Orientation

Name	Signature	Values	Default
ORIENTATION	fv	any except NaN	{ { 0.0f, 0.0f, -1.0f }, { 0.0f, 1.0f, 0.0f } }

Description: ORIENTATION is a pair of 3-tuples representing the 'at' direction vector and 'up' direction of the Object in Cartesian space. AL expects two vectors that are orthogonal to each other. These vectors are not expected to be normalized. If one or more vectors have zero length, implementation behavior is undefined. 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  Listener{n}{sifd}{v} ( enum  paramName ,  T  values );
```

4.2.3. Querying Listener Attributes

Listener state is maintained inside the AL 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  GetListener{sifd}v ( enum  param ,  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.

AL 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

AL provides calls to request and release Source Names handles. Calls to control Source Execution State are also provided.

4.3.1.1. Requesting a Source Name

The application requests a number of Sources using GenSources.

```
sizei  GenSources ( sizei  n ,  uint *  sources );
```

4.3.1.2. Releasing Source Names

The application requests deletion of a number of Sources by DeleteSources.

```
void  DeleteSources ( sizei  n ,  uint *  sources );
```

4.3.1.3. Validating a Source Name

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

```
boolean  IsSource ( uint  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.

> **Annotation (No Priorities):** There are no per Source priorities, and no explicit priority handling, defined at this point. A mechanism that lets the application express preferences in case that the implementation provides culling and prioritization mechanisms might be added at some later time. This topic is under discussion for GL as well, which already has one exlicit priority API along with internally used MRU heuristics (for resident texture memory).

4.3.2.1. Source Positioning

Table 4-5. SOURCE_RELATIVE Attribute

Name	Signature	Values	Default
SOURCE_RELATIVE	boolean	FALSE, TRUE	FALSE

SOURCE_RELATIVE set to TRUE indicates that the values specified by POSITION are to be interpreted relative to the listener position.

Annotation (Position only): SOURCE_RELATIVE does not affect velocity or orientation calculation.

4.3.2.2. Buffer Looping

Table 4-6. Source LOOPING Attribute

Name	Signature	Values	Default
LOOPING	uint	TURE, FALSE	FALSE

Description: LOOPING is a flag that indicates that the Source will not be in STOPPED state once it reaches the end of last buffer in the buffer queue. Instead, the Source will immediately promote to INITIAL and PLAYING. The default value is FALSE. LOOPING can be changed on a Source in any execution state. In particular, it can be changed on a PLAYING Source.

Annotation (Finite Repetition): Finite reptition is implemented by buffer queueing.

Annotation (Loop Control): To implement a 3 stage "loop point" solution, the application has to queue the FadeIn buffer first, then queue the buffer it wants to loop, and set LOOPING to TRUE once the FadeIn buffer has been processed and unqueued. To fade from looping, the application can queue a FadeOut buffer, then set LOOPING to false on the PLAYING source. Alternatively, the application can decide to not use the LOOPING attribute at all, and just continue to queue the buffer it wants repeated.

Annotation (Rejected alternatives): A finite loop counter was rejected because it is ambiguous with respect to persistent (initial counter) vs. transient (current counter). For similar reasons, a Play-equivalent command with a (transient) loop counter was rejected.

4.3.2.3. Current Buffer

Table 4-7. Source BUFFER Attribute

Name	Signature	Values	Default
BUFFER	ui	any valid bufferName	NONE

Description: Specify the current Buffer object, which means the head entry in its queue. Using BUFFER with the Source command on a STOPPED or INITIAL Source empties the entire queue, then appends the one Buffer specified.

For a PLAYING or PAUSED Source, using the Source command with BUFFER is an INVALID_OPERATION. It can be applied to INITIAL and STOPPED Sources only. Specifying an invalid bufferName will result in an INVALID_VALUE error while specifying an invalid sourceName results in an INVALID_NAME error.

NONE, i.e. 0, is a valid buffer Name. Source(sName, BUFFER, 0) is a legal way to release the current buffer queue on an INITIAL or STOPPED Source, whether it has just one entry (current buffer) or more. The Source(sName, BUFFER, NONE) call

still causes an INVALID_OPERATION for any source PLAYING or PAUSED, consequently it can not be abused to mute or stop a source.

Annotation (repeated Source+BUFFER does not queue): Using repeated Source(BUFFER) calls to queue a buffer on an active source would imply that there is no way to release the current buffer e.g. by setting it to 0. On the other hand read-only queues do not allow for releasing a buffer without releasing the entire queue. We can not require BUFFER state to be transient and lost as soon as a Source is implicitely or explicitely stopped. This contradicts queue state being part of the Source's configuration state that is preserved through Stop() operations and available for Play().

4.3.2.4. Queue State Queries

Table 4-8. BUFFERS_QUEUED Attribute

Name	Signature	Values	Default
BUFFERS_QUEUED	uint	[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 bufferName is 0.

Table 4-9. BUFFERS_PROCESSED Attribute

Name	Signature	Values	Default
BUFFERS_PROCESSED	uint	[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 much slots are needed for unqueueing them. On an STOPPED Source, all buffers are processed. On an INITIAL Source, no buffers are processed, all buffers are pending. This will return 0 if the current and only bufferName is 0.

Annotation (per-Source vs. Buffer State): BUFFERS_PROCESSED is only defined within the scope of a given Source's queue. It indicates that the given number of buffer names can be unqueued for this Source. It does not guarantee that the buffers can safely be deleted or refilled, as they might still be queued with other Sources. One way to keep track of this is to store, per buffer, the Source for which a given buffer was most recently scheduled (this will not work if Sources sharing buffers might be paused by the application). If necessary an explicit query for a given buffer name can be added in later revisions.

Annotation (No Looping Queues): Unqueueing requires nonzero BUFFERS_PROCESSED, which necessitates no looping on entire queues, unless we accept that no unqueueing is possible from Source looping over the entire queue. Currently not supported, as queueing is primarily meant for streaming which implies unqueue-refill-requeue operations.

4.3.2.5. Bounds on Gain

Table 4-10. Source Minimal Gain

Name	Signature	Values	Default
MIN_GAIN	f	0.0f, (0.0f, 1.0f]	0.0f

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

> **Annotation (Effective Maximal Distance):** By setting MIN_GAIN, the application implicitely defines a maximum distance for a given distance attenuation model and Source GAIN. The distance at which the effective gain is MIN_GAIN can be used as a replacement to the DirectSound3D MAX_DISTANCE parameter. Once the effective gain has reached the MIN_GAIN value, it will no longer decrease with increasing distance.

Table 4-11. Source Maximal Gain (logarithmic)

Name	Signature	Values	Default
MAX_GAIN	f	0.0f, (0.0f, 1.0f]	1.0f

Description: MAX_GAIN defines a scalar amplitude threshold. It indicates the maximal GAIN permitted for this Source. At the end of the processing of various attenuation factors such as distance based attenuation and Source GAIN, the effective gain calculated is compared to this value. If the effective gain is higher than MAX_GAIN, MAX_GAIN is applied. This happens before the Listener GAIN is applied. If the Listener gain times MAX_GAIN still exceeds the maximum gain the implementation can handle, the implementation is free to clamp. If a zero 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.

> **Annotation (Un-attenuated Source):** Setting MIN_GAIN and MAX_GAIN to the GAIN value will effectively make the Source amplitude independent of distance. The implementation is free to optimize for this situation. However, the recommended way to accomplish this effect is using a ROLLOFF_FACTOR of zero.

> **Annotation (Internal GAIN threshold):** The AL implementation is free to use an internally chosen threshold level below which a Source is ignored for mixing. Reasonable choices would set this threshold low enough so that the user will not perceive a difference. Setting MIN_GAIN for a source will override any implementation defined test.

4.3.2.6. Distance Model Attributes

Table 4-12. REFERENCE_DISTANCE Attribute

Name	Signature	Values	Default
REFER-ENCE_DISTANCE	float	[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
ROLLOFF_FACTOR	float	[0, any]	1.0f

This is used for distance attenuation calculations based on inverse distance with rolloff. For distances smaller than MAX_DISTANCE (and, depending on the distance model, larger than 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, ROLLOFF_FACTOR can be set to zero for those Sources which 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
MAX_DISTANCE	float	[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 MAX_DISTANCE will be clamped MAX_DISTANCE. MAX_DISTANCE based clamping is applied before MIN_GAIN clamping, so if the effective gain at MAX_DISTANCE is larger than MIN_GAIN, MIN_GAIN will have no effect. No culling is supported.

> **Annotation (No Culling):** This is a per-Source attribute supported for DS3D compatibility only. Other API features might suffer from side effects due to the clamping of distance (instead of e.g. clamping to an effective gain at MAX_DISTANCE).

4.3.2.7. Frequency Shift by Pitch

Table 4-15. Source PITCH Attribute

Name	Signature	Values	Default
PITCH	f	(0.0f, 1.0f]	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). Zero is not a legal value.

4.3.2.8. Direction and Cone

Each Source can be directional, depending on the settings for CONE_INNER_ANGLE and 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 GAIN is applied for the inner cone, with an application selectable CONE_OUTER_GAIN factor to define the gain in the outer zone. In the transitional zone implementation-dependent interpolation between GAIN and GAIN times CONE_OUTER_GAIN is applied.

> **Annotation (Interpolation Restrictions):** The specification does not specify the exact interpolation applied in the transitional zone, to calculate gain as a function of angle. The implementation is free to use linear or other interpolation, as long as the values are monotonically decreasing from GAIN to GAIN times CONE_OUTER_GAIN.

Table 4-16. Source DIRECTION Attribute

Name	Signature	Values	Default
DIRECTION	3fv, 3f	any except NaN	{ 0.0f, 0.0f, 0.0f }

Description: If 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 nondirectional.

> **Annotation (All Sources Directional):** From the point of view of the AL implementation, all Sources are directional. Certain choices for cone angles as well as a direction vector with zero length are treated equivalent to an omnidirectional source. The AL implementation is free to flag and optimize these cases.

Table 4-17. Source CONE_INNER_ANGLE Attribute

Name	Signature	Values	Default
CONE_INNER_ANGLE	i,f	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 omnidirectional source.

Table 4-18. Source CONE_OUTER_ANGLE Attribute

Name	Signature	Values	Default
CONE_OUTER_ANGLE	i,f	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
CONE_OUTER_GAIN	i,f	[0.0f, 1.0f]	0.0f

Description: the factor with which 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 GAIN times CONE_OUTER_GAIN. Changing GAIN affects all directions, i.e. the source is attenuated in all directions, for any position of the listener. The application has to change CONE_OUTER_GAIN as well if a different behavior is desired.

Annotation (GAIN calculation): The angle-dependend gain DGAIN is multiplied with the gain determined by the source's GAIN and any distance attenuation as applicable. Let theta be the angle between the source's direction vector, and the vector connection the source and the listener. This multiplier DGAIN is calculated as:

```
OUTER = CONE_OUTER_ANGLE/2;
INNER = CONE_INNER_ANGLE/2;
if      ( theta less/equal INNER )
    DGAIN = 1
else if ( theta greater/equal OUTER )
   DGAIN = CONE_OUTER_GAIN
else
   DGAIN = 1 - (1-CONE_OUTER_GAIN)*((theta-INNER)/(OUTER-INNER))
GAIN *= DGAIN
```

in the case of linear interpolation. The implementation is free to use a different interplation across the (INNER,OUTER) range as long as it is monotone.

Annotation (CONE_OUTER_GAIN always less than GAIN): CONE_OUTER_GAIN is not an absolute value, but (like all GAIN parameters) a scaling factor. This avoids a possible error case (implementations can count on effective gain outside the outer cone being smaller than GAIN), and ensures the common case in which changing GAIN should affect inner, transitional, and outer zone simultaneously.

In case that the application desires to have an outer zone volume exceeding that of the inner cone, the mapping to AL will require to rotate the Source direction to the opposite direction (negate vector), and swapping inner and outer angle.

4.3.3. Changing Source Attributes

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

```
void Source{n}{sifd} ( uint sourceName , enum paramName , T
value );
```

```
void Source{n}{sifd}v ( uint sourceName , enum paramName , T *
values );
```

4.3.4. Querying Source Attributes

Source state is maintained inside the AL 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 Source*.

```
void GetSource{n}{sifd}{v} ( uint sourceName , enum paramName , T
* values );
```

4.3.5. Queueing Buffers with a Source

AL does not specify a built-in streaming mechanism. There is no mechanism to stream data e.g. into a Buffer object. Instead, the API introduces 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, whether these are re-used in cycle, pooled, or thrown away.
- Looping (over a finite number of repititions) can be implemented by explicitely 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, queueing 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.

4.3.5.1. Queueing command

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

```
void alSourceQueueBuffers ( uint sourceName , sizei numBuffers ,
uint * bufferNames );
```

This command is legal on a Source in any state (to allow for streaming, queueing has to be possible on a PLAYING Source). Queues are read-only with exception of the unqueue operation. The Buffer Name NONE (i.e. 0) can be queued.

Annotation (BUFFER vs. SourceQueueBuffers): A Sourcei(sname, BUFFER, bname) command is an immediate command, and executed immediately. It effectively unqueues all buffers, and then adds the specified buffer to the then empty queue as its single entry. Consequently, this call is only legal if SourceUnqueueBuffers is legal. In particular, the Source has to be STOPPED or INITIAL. The application is still obliged to delete all buffers as were contained in the queue. Sourcei(sname, BUFFER, NONE) is a legal command, effectively wiping the queue without specifying an actually playable buffer.

Annotation (Buffer Repetition): To accomplish a finite number of repetitions of a buffer name multiple times, the buffer has to be queued multiple times. If the need occurs, the API could be extended by SourceQueueBuffer(sname, bname, repetitions) call for brevity.

Annotation (Backwards Compatiblity): Sourcei(sname, BUFFER, bname) has been rejected as a queueing command, as it would make semantics dependent on source state (queueing if PLAYING, immediate else). The command is not legal on a PLAYING or PAUSED Source.

Annotation (No BUFFER_QUEUE): Duplication of one entry point is preferable to duplicating token enums, and tokens do not express commands, but specify the attribute/state affected. From the same reason, there is no BUFFER_UNQUEUE token-as-command.

4.3.5.2. Unqueueing 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 (PLAYING or PAUSED Source).

The Unqueue command removes a number of buffers entries that have finished processing, in the order of appearance, from the queue. The operation will fail if more buffers are requested than available, leaving the destination arguments unchanged. An INVALID_VALUE error will be thrown. If no error, the destination argument will have been updated accordingly.

```
void SourceUnqueueBuffers ( uint sourceName , sizei numEntries ,
uint * bufferNames );
```

Annotation (Unqueueing shared buffers): If a buffer is queued with more than one source, it might have been processed for some not all of them. With the current interface, the application is forced to maintain its own list of consumers (Sources) for a buffer it wishes to unqueue. For groups of Sources that are never individually PAUSED nor STOPPED, the application can save the MRU Source for which the buffer was scheduled last.

Annotation (Looping a Queue vs. Unqueue):: If a Source is playing repeatedly, it will traverse the entire Queue repeatedly. Consequently, no buffer in the queue can be considered processed until there is no further repetition scheduled.

Annotation (No Name based access): No interface is provided to access a queue entry by name, due to ambiguity (same buffer name scheduled several times in a sequence).

Annotation (No Index based access): No interface is provided for random access to a queue entry by index.

4.3.5.3. More Annotation on Queueing

Annotation (No Queue Copying): The current queue of a source could be copied to another source, as repetition and traversal parameters are stored unless the queue entry is unqueued, or the queue is replaced using AL_BUFFER. Copying a queue is a special case of copying Source state in one sense, and a special case of a synching problem in another. Due to these unresolved issues no such command is included in the current specification. To share queues, the application can keep buffer names and the selected attributes that define the queue entries in an array or other lookup table.

Annotation (No Explicit QueueClear): Sourcei(sname, BUFFER, NONE) serves the same purpose. The operation is also redundant with respect to Unqueue for a STOPPED Source.

Annotation (Queueing vs. AppendData):: Buffer queueing does not solve the synchronization and timing issues raised by possible underflow, as these are inherent to application-driven (pushed) streaming. However, it turns an internal AL error condition (offset exceeds valid data) into an audible artifact (Source stops). Its main advantage is that it allows the application coder to operate at a scale of her own choice, selecting the number and size of buffers used for caching the stream, and to schedule buffer refill and queueing according to preferences and constraints. Queueing effectively moves all problems related to replacing or appending Buffer data to the scale of entire arrays istead of single samples and indices.

Annotation (Multiple Sources on a stream): Queueing allows for the application to determine how much of a backlog of the data stream is preserved. The application can keep buffers, and queue them with other Sources after they have been used already by the original Source. Unlike the mechanism for appending data to a buffer, the backlog is visible to the application and under its control, and no synchronization of Sources using the stream is required.

Annotation (Loop Points and Compressed Data): For compressed data, uncompression by the application might be impossible or undesireable. In consequence, splitting the sample into several buffers is not possible without explicit support by the API. Buffer-Buffer operations will be added as needed, for the time being applications should not try to use compressed samples if more than full looping is required.

Annotation (No Explicit Queue Objects): Explicit Queue objects have been considered and rejected, as they introduce another producer-consumer dependency with another level of indirection. Further, e.g. QUEUE would also require deprecating BUFFER (breaking backwards compatibility) as an alSource argument, or would introduce a confusing set of precedence and overide rules if both are used in sequence. However, in the absence of explicit queue objects the application will be forced to keep track where buffers have been queued in case it intends to unqueue them for refill or deletion. If several sources use the same buffers (e.g. for synchronous or asynchronous streaming) the buffer will have to be unqueued from each single one.

4.3.6. Managing Source Execution

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

TBA: State Transition Diagram.

Annotation/ Source Config/Exec State: Sources have configuration state and execution state. Configuration state is directly set by the application using AL commands, starting with the INITIAL configuration. Execution state (e.g. the offset to the current sample) is not under direct application control and not exposed.

4.3.6.1. Source State Query

The application can query the current state of any Source using GetSource with the parameter Name SOURCE_STATE. Each Source can be in one of four possible execution states: INITIAL, PLAYING, PAUSED, STOPPED. Sources that are either PLAYING or PAUSED are considered active. Sources that are STOPPED or INITIAL are considered inactive. Only 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.

4.3.6.2. 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  SourcePlay ( uint  sName );

void  SourcePause ( uint  sName );
```

```
void  SourceStop ( uint  sName );

void  SourceRewind ( uint  sName );
```

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

```
void  SourcePlayv ( sizei  n ,  uint *  sNames );

void  SourcePausev ( sizei  n ,  uint *  sNames );

void  SourceStopv ( sizei  n ,  uint *  sNames );

void  SourceRewindv ( sizei  n ,  uint *  sNames );
```

The following state / command / state transitions are defined:

- Play() applied to an INITIAL Source will promote the Source to PLAYING, thus the data found in the Buffer will be fed into the processing, starting at the beginning. Play() applied to a PLAYING Source will restart the Source from the beginning. It will not affect the configuration, and will leave the Source in PLAYING state, but reset the sampling offset to the beginning. Play() applied to a PAUSED Source will resume processing using the Source state as preserved at the Pause() operation. Play() applied to a STOPPED Source will propagate it to INITIAL then to PLAYING immediately.
- Pause() applied to an INITIAL Source is a legal NOP. Pause() applied to a PLAYING Source will change its state to PAUSED. The Source is exempt from processing, its current state is preserved. Pause() applied to a PAUSED Source is a legal NOP. Pause() applied to a STOPPED Source is a legal NOP.
- Stop() applied to an INITIAL Source is a legal NOP. Stop() applied to a PLAYING Source will change its state to STOPPED. The Source is exempt from processing, its current state is preserved. Stop() applied to a PAUSED Source will change its state to STOPPED, with the same consequences as on a PLAYING Source. Stop() applied to a STOPPED Source is a legal NOP.
- Rewind() applied to an INITIAL Source is a legal NOP. Rewind() applied to a PLAYING Source will change its state to STOPPED then 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. Rewind() applied to a PAUSED Source will change its state to INITIAL, with the same consequences as on a PLAYING Source. Rewind() applied to a STOPPED Source promotes the Source to INITIAL, resetting the sampling offset to the beginning.

Annotation (SourceNext): The specification does not provide any means to immediately skip from the current Buffer to the next in the queue. A conditional stop (following the next complete traversal) is available. If necessary an additonal entry point could be provided in future revisions.

Annotation (Rewind() optional): The INITIAL state is not identical to the STOPPED state. Applications that want to verify whether a Source has indeed been PLAYING

before becoming STOPPED can use Rewind() to reset the Source state to INITIAL. This is an optional operation that can safely be omitted by application without this constraint. Applications that want to guard against Play() on a Source that is INITIAL can query the Source state first.

Annotation (Play() on a PLAYING Source): Repeated Play() commands applied a PLAYING Source are interpreted as an (atomic) sequence to stop and restart a Source. This can be used by applications that want to restart a sound but do not care whether the Source has finished or not, and do not want an audible pause. One example is the DOOM chaingun repeatedly abbreviating the pistol sound. To guard against redundant Play() commands, an application can query the current state before executing Play(). If the application coder wants to be sure that the Source will play the buffer again, she can either increment PLAY_COUNT, or queue the buffer.

Annotation (redundant commands): The simple variant (e.g. SourcePlay) is redundant to the vector variant (e.g. SourcePlayv). However, these calls will be used frequently, and the simple variant is provided for convenience. However, AL does not enable applications to use literals as source names.

4.3.6.3. Resetting Configuration

The INITIAL state is not necessarily identical to the default state in which Source is created. INITIAL merely indicates that the Source can be executed using the SourcePlay command. A STOPPED or INITIAL Source can be reset into the default configuration by using a sequence 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.

Annotation (illegal NOPs): In the current specification there are no illegal NOPs. In other words, no sequence of commands affecting the execution state will generate an INVALID_OPERATION error.

Chapter 5. Buffers

A Buffer encapsulates AL 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 Sources and Listener, 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 driver and hardware to optimize storage and processing where applicable.

The simplest supported format for buffer data is PCM.

> **Annotation/ Compressed Buffers:** Compressed formats are in no way guaranteed by the implementation to remain compressed. The driver might have to uncompres in memory at once, if no hardware-assisted or incremental decoding is possible. In many cases an implementation has to decompress the buffer, converting the uncompressed data to a canonical internal format, and resample it into the format native to the current context.

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 no 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 which have dereferenced the data in the Buffer in its entirety, and can not 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 STOPPED or INITIAL 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 queueing 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 BufferData commands.

> **Annotation (No CURRENT State):** For buffer queueing, it is not relevant whether the Buffer data is currently dereferenced by any Source or not. It is therefore not necessary

to distinguish a CURRENT state (being referenced as current buffer by a single PLAYING or PAUSED Source).

Annotation (State Query and Shared Buffers): A buffer that is unused by one Source might be used by another. The Unqueue operation is determined by the number of queue entries already processed by the given Source. However, the application has to check whether the Buffer is still in use by other Sources. For now, applications have to maintain their own lists of buffer consumer (source) lists. If necessary, an explicit call to determine current buffer state with respect to all Sources might be added in future revisions.

5.2. Managing Buffer Names

AL 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 GenBuffers.

```
void  GenBuffers ( sizei  n ,  uint *  bufferNames );
```

5.2.2. Releasing Buffer Names

The application requests deletion of a number of Buffers by calling DeleteBuffers.

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

```
void  DeleteBuffers ( sizei  n ,  uint *  bufferNames );
```

IsBuffer(bname) can be used to verify deletion of a buffer. Deleting bufferName 0 is a legal NOP in both scalar and vector forms of the command. The same is true for unused buffer names, e.g. such as not allocated yet, or as released already.

5.2.3. Validating a Buffer Name

The application can verify whether a buffer Name is valid using the IsBuffer query.

```
boolean  IsBuffer ( uint  bufferName);
```

5.3. Manipulating Buffer Attributes

5.3.1. Buffer Attributes

This section lists the attributes that can be set, or queried, per Buffer. Note that some of these attributes can not be set using the Buffer commands, but are set using commands like BufferData.

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

Table 5-1. Buffer FREQUENCY Attribute

Name	Signature	Values	Default
FREQUENCY	float	none	(0, any]

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

Annotation (No Frequency enumeration): As the implementation has to support conversion from one frequency to another to implement pitch, it is feasible to offer support for arbitrary sample frequencies, instead of restricting the application to an enumeration of supported sample frequencies. Another reason not to limit frequency to an enumerated set is that future hardware might support variable frequencies as well (it might be preferable to choose the sampling frequency according to the PSD of the signal then).

However, it is desirable to avoid conversions due to differences between the sample frequency used in the original data, the frequency supported during the mixing, and the frequency expected by the output device.

Annotation (Implied Frequency): To account for the possibility of future AL implementations supporting encoding formats for the application might not want, or be able, to retrieve the actual frequency from the encoded sample, the specification will be amended to guarantee the following behavior: If a nonzero frequency is specified, it will force a conversion from the actual to the requested frequency. If the application specifies a 0 frequency, AL will use the actual frequency. If there is no frequency information implied by the format or contained in the encoded data, specifying a 0 frequency will yield INVALID_VALUE. It is recommended that applications use NONE instead of the literal value.

Annotation (No Format query): As of this time there is no query for FORMAT, or format related state information. Query of the channels or bits of a given buffer make little sense if the query the internal (canonical, not buffer specific) format. Query of the original sample data format makes little sense unless the implementation is obliged to preserve the original data.

Table 5-2. Buffer SIZE Attribute

Name	Signature	Values	Default
SIZE	sizei	[0, MAX_UINT]	0

Description: Size in bytes of the buffer data. Query through GetBuffer, can be set only using BufferData calls. Setting a SIZE of 0 is a legal NOP. The number of bytes does not necessarily equal the number of samples (e.g. for compressed data).

5.3.2. Querying Buffer Attributes

Buffer state is maintained inside the AL implementation and can be queried in full. The valid values for paramName are identical to the ones for Buffer*.

```
void GetBuffer{n}{sifd}{v} ( uint bufferName, enum paramName , T *
values );
```

5.3.3. 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 BufferData.

```
void BufferData{n}{sifd}{v} ( uint bufferName, enum format, void *;
data , sizei size , uint 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. The internal format of the Buffer is not exposed to the application, and not accessible. Valid formats are FORMAT_MONO8, FORMAT_MONO16, FORMAT_STEREO8, and FORMAT_STEREO16. An implementation may expose other formats, see the chapter on Extensions for information on determining if additional formats are supported.

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

Chapter 6. AL Contexts and the ALC API

This section of the AL specification describes ALC, the AL Context API. ALC is a portable API for managing AL 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 intance of the AL state machine.

To avoid confusion with the AL related prefixes implied throughout this document, the "alc" and "ALC_" prefixes have been made explicit in the ALC related sections.

ALC defines the following objects: Contexts.

Annotation (ALC entry points): While the actual ALC implemention might be supplied as a separate library, or as part of a server or daemon, the specification requires that the AL library provides the actual ALC entry points.

Annotation (ALC OS independend): ALC is meant to be OS-independent. OS specifics are expected to be adressed by defining proper device specifiers strings, and configuration attributes. In this, ALC differs from GLX/WGL, which (due to the tighter coupling with the window manager and operating system) attempt to abstract OS specifics to a much lesser degree.

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 uniq7ue device specifiers (represented as strings).

Annotation (Network transparency): AL is meant for interoperability with OpenGL. Some implementations of OpenGL bindings (e.g. GLX) are network transparent. The Device API theoretically allows for a network transparent AL implementation. No wire protocol is specified, no specification or implementation is planned.

Annotation (Device Enumeration): At this time, ALC does not provide mechanism to query for available devices, and request device enumerations. This might be added at a later time, depending on demand and the ability to abstract OS and configuration specifics.

Annotation (X11 Audio): The ALC API intentionally mimicks XOpenDisplay and XCloseDisplay. There is no X Audio standard, although proposals have been made in the past. The ALC API design accounts for this possibility in a minimal way.

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 ubyte * 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.

> **Annotation (Operating system dependencies):** At this point, system specific configuration, and operating system specific details, are handled by leaving the details of the string specifier to the implementation. The application coder has to determine how he wants to obtain this information from the OS or the user. If, at a later point, device enumeration and configuration requests are supported through ALC, the resulting string might still be operating system and implementation specific.

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).

```
void alcCloseDevice( ALCdevice * deviceHandle);
```

If deviceHandle is NULL or invalid, an ALC_INVALID_DEVICE error will be generated. Once closed, a deviceHandle is invalid.

6.2. Managing Rendering Contexts

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

The default AL Context interoperates with a hardware device driver. The application manages hardware and driver resources by communicating through the ALC API, and configures and uses such Contexts by issuing AL API calls. A default AL Context processes AL calls and sound data to generate sound output. Such a Context is called a Rendering Context. There might be non-rendering contexts in the future.

The word "rendering" was chosen intentionally to emphasize the primary objective of the AL API - spatialized sound - and the underlying concept of AL as a sound synthesis pipeline that simulates sound propagation by specifying spatial arrangements of listeners, filters, and sources. If used in describing an application that uses both OpenGL and AL, "sound rendering context" and "graphics rendering context" should be used for clarity. Throughout this document, "rendering" is used to describe spatialized audio synthesis (avoiding ambiguous words like "processing", as well as proprietary and restrictive terms like "wavetracing").

6.2.1. Context Attributes

The application can choose to specify certain attributes for a context. Attributes not specified explicitely are set to implementation dependend defaults.

Table 6-1. Context Attributes

Name	Description
ALC_FREQUENCY	Frequency for mixing output buffer, in units of Hz.
ALC_REFRESH	Refresh intervalls, in units of Hz.
ALC_SYNC	Flag, indicating a synchronous context.

Annotation (Refresh Control): Applications might have a fixed, or bounded, schedule for state changes (e.g. synchronously with the GL framerate). In this case it is desirable to specify the mixahead interval (milliseconds), or refresh rate (Hz), for the mixing thread. This is especially important for a synchronous context, where the application has to specify the refresh interval it intends to keep.

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 ,
int *  attrList );
```

Context creation will fail if the application requests attributes that, by themselves, can not be provided. Context creation will fail if the combination of specified attributes can not be provided. Context creation will fail if a specified attribute, or the combination of attributes, does not match the default values for unspecified attributes.

6.2.3. Selecting a Context for Operation

To make a Context current with respect to AL Operation (state changes by issueing commands), alcMakeContextCurrent is used. The context parameter can be NULL or a valid context pointer. The operation will apply to the device that the context was created for.

```
boolean  alcMakeContextCurrent ( ALCcontext *  context );
```

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.

Annotation (No Explicit Device): An ALC context is bound to the device it was created for. The context carries this information, thus removing the need to specify the device

explicitely. Contexts can not be made current for any other device aside from the one they were created for.

Annotation (No Multiple Current): There is only one current context per process, even in multithreaded applications, even if multiple devices are used.

Annotation (Current NULL): The implementation is encouraged to exploit optimizations possible if the application sets the current context to NULL, indicating that no state changes are intended for the time being. The application should not set the current context to NULL if more state changes are pending on the most recent, or another context created for the same device.

Annotation (Shared Objects): Buffers are shared among contexts. As mutiple contexts can exist at the same time, the state of shared objects is also shared among 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 like offset increments are supposed to be performed), the application has to use 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 not marked as processing.

Annotation (Sync and async implementations): Unfortunately, the exact semantics of alcProcessContext is not independend of the implementation. Ideally it should be completely transparent to the application whether the sound driver is threaded or synced. Unfortunately a synced context has to have its execution initiated by the application, which requires calls of alcProcessContext timed in accordance to the drivers mixahead, or the rendering buffer will underflow. For a threaded driver, the implementation is free to consider alcProcessContext a NOP once the context has been marked as processing.

One consequence is that an application that was developed using a threaded implementation of AL might not work properly with a synchronous implementation of AL (on the other hand, an AL application that works using a synchronous implementation is guaranteed to work with a threaded implementation.

Enforcing alcProcessContext calls would defeat the purpose of a threaded implementation. Permitting the AL implementation to e.g. schedule optimizations based on alcProcessContext calls would similarly obfuscate the exact semantincs. Consequently, the application coder has to accept this implementation dependency, and has to rely on the ALC_SYNC attribute to explicitely request a synchronous implementation. The implementation can expect the application to be aware of the additonal constraints imposed on alcProcessContext in this case.

Annotation (Multiple Contexts and SYNC refresh): The application can request SYNC contexts or threaded contexts, however, the implementation is not obliged to provide both, or provide a mixture of both on the same device.

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 like offset increments is not supposed to be changed), the application has to use alcSuspendContext.

```
void alcSuspendContext( ALCcontext *  context );
```

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

Annotation (Sync and async implementations): Unfortunately, the exact semantics of alcSuspendContext is also not independend of the implementation. For a threaded implementation, alcSuspendContext is necessary to ensure a context is not processed. For a synchronous implementation, omitting alcProcessContext calls will ultimately have the same effect, but will also generate rendering buffer underflow errors. Again, the application coder that requests a synchronous context using ALC_SYNC has to make sure that alcSuspendContext is used accordingly.

Annotation (Suspending vs. Muting a context): By setting Listener GAIN to zero, an application can mute a context, and expect the implementation to bypass all rendering. However, the context is still processing, and the internal execution state is still updated accordingly. Suspending a context, whether muted or not, will incidentally suspend rendering as well. However, it is the application's responsibility to prevent artifacts (e.g. by proper GAIN control to fade in and out). It is recommended to mute a context before suspending.

Annotation (Current Context Suspended): It is possible to make a suspended context current, or suspend the current context. In this case, the implementation is still obliged to immediately verify AL commands as they are issued, and generate errors accordingly. The implementation is permitted to postpone propagating the actual state changes until the context is marked for processing again, with the exception of dereferencing data (e.g. buffer contents). For efficiency reasons (memory usage), most if not all AL commands applied to a suspended context will usually be applied immediately. State changes will have to be applied in the sequence they were requested. It is possible to use suspension of a current context as an explicit locking (to enforce apparent synchronicity), but execution is still guaranteed to be in sequence, and the implementation is not expected to optimize this operation. A typical use would be setting up the initial configuration while loading a scene.

Annotation (Release of Hardware Resources): The specification does not guarantee that the implementation will release hardware resources used by a suspended context. This might well depend on the details of the hardware and driver. Neither a muted context nor a suspended context can be expected to free device resources. If all contexts for a given device are suspended, and no context of this device is current, the implementation is expected to release all hardware resources if possible.

6.2.6. Destroying a Context

```
void  alcDestroyContext ( ALCcontext *  context );
```

The correct way to destroy a context is to first release it using alcMakeCurrent and NULL. Applications should not attempt to destroy a current context.

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

```
boolean IsExtensionPresent( const ALCdevice * deviceHandle, const
ubyte *  extName );
```

A NULL name argument returns FALSE, as do invalid and unsupported string tokens. A NULL deviceHandle will result in an INVALID_DEVICE error.

Annotation (Exlicit Device Parameter): Certain ALC Extensions might be relevant to context creation (like additional attributes, or support for unusual multi-context combinations), thus the application might have to query these before a context is created. On the other hand, ALC Extensions are specific to the device.

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.

```
void *  alcGetProcAddress( const ALCdevice * deviceHandle,  const
ubyte *  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 independend, 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 AL core are guaranteed. Availability of extension tokens dependends on the ALC extension.

```
uint  alcGetEnumValue ( const ALCdevice * deviceHandle,  const ubyte
enumName );
```

Specifying a NULL name parameter will cause an ALC_INVALID_VALUE error.

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.

```
enum alcGetError( ALCdevice * deviceHandle);
```

Error conditions are specific to the device.

Table 6-2. Error Conditions

Name	Description
ALC_NO_ERROR	The device handle or specifier does name an accessible driver / server.
ALC_INVALID_DEVICE	The Context argument does not name a valid context.
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	An value (e.g. attribute) is not valid, or not applicable.

Annotation (No UNDERFLOW error): Applications using synchronous (and, depending on CPU load, even an asynchronous implementation itself) might fail to prevent

underflow of the rendering output buffer. No ALC error is generated in these cases, as it this error condition can not be applied to a specific command.

6.3.7. String Query

The application can obtain certain strings from ALC.

```
const ubyte * alcGetString( ALCdevice * deviceHandle,  enum  token );
```

For some tokens, NULL is is a legal value for the deviceHandle. In other cases, specifying a NULL device will generate an ALC_INVALID_DEVICE error.

Table 6-3. String Query Tokens

Name	Description
ALC_DEFAULT_DEVICE_SPECIFIER	The specifier string for the default device (NULL handle is legal).
ALC_DEVICE_SPECIFIER	The specifier string for the device (NULL handle is not legal).
ALC_EXTENSIONS	The extensions string for diagnostics and printing.

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,  enum  token ,  sizei
size ,  int  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. A NULL destination or a zero size parameter will cause ALC to ignore the query.

Table 6-4. Integer Query Tokens

Name	Description
ALC_MAJOR_VERSION	Major version query.
ALC_MINOR_VERSION	Minor version query.

Name	Description
ALC_ATTRIBUTES_SIZE	The size required for the zero-terminated attributes list, for the current context. NULL is an invalid device. NULL (no current context for the specified device) is legal.
ALC_ALL_ATTRIBUTES	Expects a destination of ALC_CURRENT_ATTRIBUTES_SIZE, and provides the attribute list for the current context of the specified device. NULL is an invalid device. NULL (no current context for the specified device) will return the default attributes defined by the specified device.

Annotation (Backward Compatibility): Backward compatibility is guaranteed only for minor revisions. Breaking ABI backwards compatibility will require a issuing major revision.

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 impose by all contexts, not just the current context. This includes suspended contexts as well as contexts that are processing.

Appendix A. Global Constants

Table A-1. Misc. AL Global Constants

Name	OpenAL: datatype	Description	Literal value
ALenum	FALSE	boolean false	0
ALenum	TRUE	boolean true	1

Appendix B. Extensions

Extensions are a way to provide for future expansion of the AL 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. The handling of vendors-specific or multi-vendor extensions is left to the implementation. The IA-SIG I3DL2 Extension is an example of multi-vender extensions to the current AL core API.

B.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 will have to verify the presence of an extension using IsExtensionPresent(). The application can then retrieve the address (function pointer) of an extension entry point using GetProcAddress. Extensions and entry points can be Context-specific, and the application can not count on an Extension being available based on the mere return of an entry point. The application also has to maintain pointers on a per-Context basis.

```
boolean IsExtensionPresent( const ubyte * extName);
```

Returns TRUE if the given extension is supported for the current context, FALSE otherwise.

Annotation (IsExtensionPresent): This function is inspired by the GLU addition, but placed in the core API as we intend to avoid a separate ALU. This function avoids potential string overflow and string parsing issues (strstr) raised by GetString(EXTENSIONS).

Annotation/ EXTENSIONS: GetString(EXTENSIONS) is supported as well, as it allows for easy archiving and priting of the list of supported extensions.

B.2. Retrieving Function Entry Addresses

```
void * GetProcAddress( const ubyte * 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 GetProcAddress to obtain core API entry points, not just extensions. This is the recommended way to dynamically load and unload AL DLL's as sound drivers.

B.3. Retrieving Enumeration Values

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

```
uint  GetEnumValue ( const ubyte  enumName );
```

Returns 0 if the enumeration can not be found. 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. Implementations should not attempt to return 0 to indicate that the extensions is not supported for the current context.

> **Annotation/ enums with value zero:** The literal value 0 is guaranteed for a number of AL enums, such as FALSE, NONE, ZERO. As with GL applications might employ sloppy use of this identity. It also means that enums with zero value can not be queried through GetEnumValue, a minor flaw given the constraints of ABI backward compatibility. The recommended value to compare GetEnumValue results with is NONE.

B.4. Naming Conventions

Extensions are required to use a postfix that separates the extension namespace from the core API's namespace. For example, an ARB-approved extension would use "_ARB" with tokens (ALenum), and "ARB" with commands (function names). A vendor specific extension uses a vendor-chosen postfix, e.g. Loki Extensions use "_LOKI" and "LOKI", respectively.

B.5. ARB Extensions

There are no ARB Extensions defined yet, as the ARB has yet to be installed.

B.6. Other Extension

For the time being this section will list externally proposed extensions, namely the extension based on the IASIG Level 2 guideline.

B.6.1. IA-SIG I3DL2 Extension

The IA-SIG I3DL2 guideline defines a set of parameters to control the reverberation characteristics of the environment the listener is located in, as well as filtering or muffling effects applied to individual Sources (useful for simulating the effects of obstacles and partitions). These features are supported by a vendor neutral extension to AL (TBA). The IA-SIG 3D Level 2 rendering guideline[1] provides related information.

B.7. Compatibility Extensions

The extensions described have at one point been in use for experimental purposes, proof of concept, or short term needs. They are preserved for backwards compatibility. Use is not recommended, avialability not guaranteed. Most of these will be officially dropped by the time API revision 2.0 is released.

B.7.1. Loki Buffer InternalFormat Extension

AL currently does not provide a separate processing chain for multichannel data. To handle stereo samples, the following alternative entry point to BufferData has been defined.

```
void  BufferWriteData( uint bufferName,  enum format,  void *;  data ,
 sizei  size ,  uint frequency,  enum internalFormat);
```

Valid formats for internalFormat are FORMAT_MONO8, FORMAT_MONO16, FORMAT_STEREO8, and FORMAT_STEREO16.

B.7.2. Loki BufferAppendData Extension

Experimental implementation to append data to an existing buffer. Obsoleted by Buffer Queueing. TBA.

> **Annotation (GenStreamingBuffers)::** It is possible that a consistent implementation of this extension will require distinguishing streaming from regular buffers at creation time, instead of making this distinction implied by the use of BufferData vs. BufferAppendData.

B.7.3. Loki Decoding Callback Extension

Experimental implementation to allow the application to specify a decoding callback for compression formats and codecs not supported by AL. This is supposed to be used if full uncompression by the application is prohibited by memory footprint, but streaming (by queueing) is not desired as the compressed data can be kept in memory in its entirety.

If mixing can be done from the compressed data directly, several sources can use the sample without having to be synchronized. For compression formats not supported by AL, however, partial decompression has to be done by the application. This extension allows for the implementation to "pull" data, using apllication provided decompression code.

The use of this callback by the AL implementation makes sense only if late decompression (incremerntal, on demand, as needed for mixing) is done, as full early compression (ahead-of-time) inside the implementation would exact a similar memory footprint.

TBA.

This extension forces execution of third party code during (possibly threaded) driver operation, and might also require state management with global variables for decoder state, which raises issues of thread safety and use for multiple buffers. This extension should be obsolete as soon as AL supports a reasonable set of state of the art compression and encoding schemes.

B.7.4. Loki Infinite Loop Extension

To support infinite looping, a boolean LOOP was introduced. With the introduction of buffer queueing and the request for support for a limited number of repetitions, this mechanism was redundant. This extension is not supported for buffer queue operations, attempts to use it will cause an ILLEGAL_OPERATION error. For backwards compatibility it is supported as the equivalent to

```
Source( sName, PLAY_COUNT, MAX_INTEGER )
```

For the query LOOP==TRUE, the comparison PLAY_COUNT!=MAX_INTEGER has to be executed on the queue, not the current value which is decremented for a PLAYING Source.

Table B-1. Source LOOP_LOKI Attribute

Name	Signature	Values	Default
LOOP_LOKI	b	TRUE FALSE	FALSE

Description: TRUE indicates that the Source will perform an inifinite loop over the content of the current Buffer it refers to.

B.7.5. Loki Byte Offset Extension

The following has been obsoleted by explicit Source State query. hack.

Table B-2. Buffer BYTE Offset attribute

Name	Signature	Values	Default
BYTE_LOKI	ui	n/a	n/a

Current byte for the buffer bound to the source interpreted as an offset from the beginning of the buffer.

B.8. Loop Point Extension

In external file now.

Notes

1. http://www.iasig.org/pages/wg/3DWG/3dwg.htm

Appendix C. 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 AL discussion list (please use RFC in the Subject line), or send to the maintainer of the AL specification. If you are shipping an actual implementation as a patch or as part of the AL CVS a formal writeup is recommend. In this case, the Extension has to be described as part of the specification, which is maintained in DocBook SGML (available for UNIX, Linux and Win32). The SGML source of the specification is available by CVS, and the Appendix on Extensions can be used as a template. Contact the maintainer for details.