OpenSL ES Specification

Version 1.1

January 18th 2011

Table of Contents

PART 1: USER MANUAL

1 Overview

1.1 Purpose of this Document

This document specifies the Application Programming Interface (API) for OpenSL ES™ 1.1. Developed as an open standard by the Khronos Group, OpenSL ES is a native language application-level audio API for embedded and mobile multimedia devices. It provides a device-independent cross-platform interface for applications to access a device's audio capabilities.

1.1.1 About the Khronos Group

The Khronos Group is a member-funded industry consortium focused on the creation of open standard, royalty-free APIs to enable the authoring and accelerated playback of dynamic media on a wide variety of platforms and devices. All Khronos members can contribute to the development of Khronos API specifications, are empowered to vote at various stages before public deployment, and may accelerate the delivery of their multimedia platforms and applications through early access to specification drafts and conformance tests. The Khronos Group is responsible for open APIs such as OpenGL® ES, OpenMAX™ and OpenVG™.

1.2 Scope

OpenSL ES is an application-level C-language audio API designed for resource-constrained devices. Particular emphasis has been made to ensure that the API is suitable for mobile and embedded devices, including basic mobile phones, smart phones, tablets and portable music players. The API is suitable for both lower-end devices and feature-rich, higher-end devices.

OpenSL ES's API design devotes particular attention to application-developer friendliness. Its status as an open cross-platform API enables developers to port the same source across multiple devices with minimal effort. Thus, OpenSL ES provides a stable base for application development.

This document specifies the OpenSL ES API. It does not define or suggest how to implement the API.

1.3 Intended Audience

This specification is meant for application-developers and implementers. The document is split into two sections: a user guide and an API reference. Application-developers can use this document as a user guide to learn about how to use OpenSL ES and they can refer to the API reference when developing their applications. Implementers of the API can use this specification to determine what constitutes an implementation conforming to the OpenSL ES standard.

1.4 A Brief History of OpenSL ES

OpenSL ES was originally conceived with a specific goal of reducing the API fragmentation problems that confront audio applications on mobile devices. Prior to the release of OpenSL ES there were many (mostly proprietary) audio APIs. As a result any developer wanting to deliver their audio application on many devices spent most of their time porting an application, as opposed to maximizing features and functionality. Even when an audio API was supported on multiple devices, there was little guarantee as to what level of functionality could be relied on between the devices. It was clear that as more advanced audio functionality became commonplace on mobile devices, this fragmentation would only worsen.

This situation inspired companies with a wide range of backgrounds – such as mobile phone handset manufacturers, PC audio hardware manufacturers, MIDI vendors, audio silicon vendors and audio software vendors - to create OpenSL ES. An open standard, C-language audio API for mobile embedded devices, OpenSL ES provides a guarantee of what functionality will be supported in all implementations of the API.

1.5 Relationship to OpenMAX AL

OpenSL ES is an application-level enhanced audio API, specifically designed for mobile embedded devices. OpenMAX AL [OMXAL], also from the Khronos Group, is an application-level multimedia playback and recording API for mobile embedded devices. As such, both APIs do overlap in certain basic audio functionality (such as audio playback, audio recording and basic MIDI). The Venn diagram in Figure 1 illustrates the functional overlap in the two APIs.

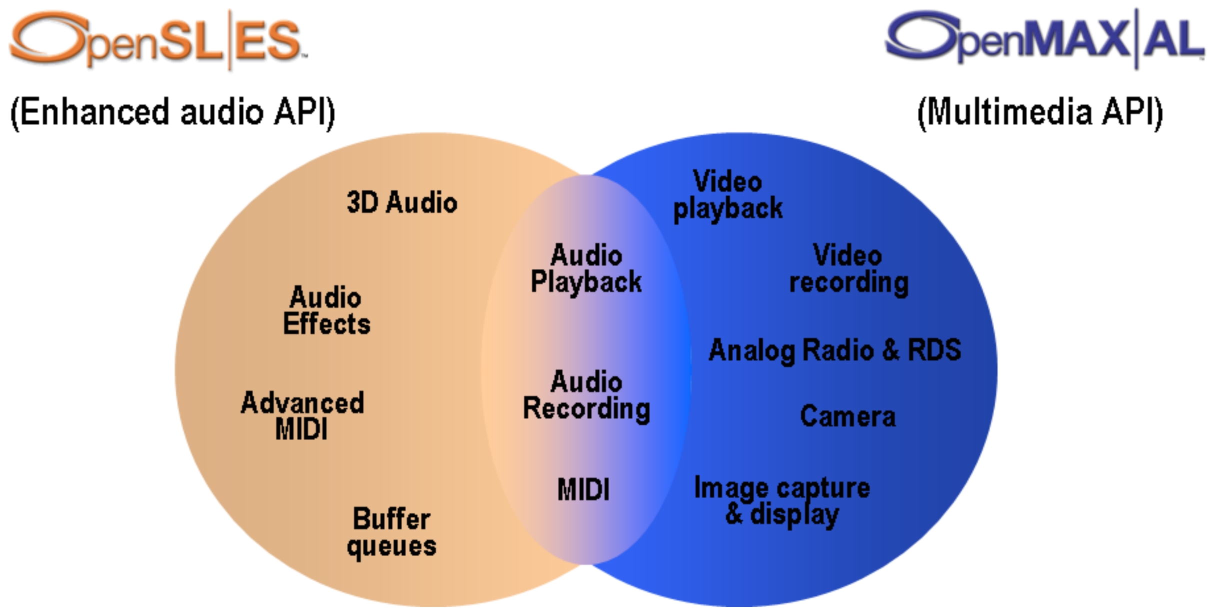


Figure 1: OpenSL ES versus OpenMAX AL

As the Venn diagram shows, OpenSL ES has advanced audio features such as effects (reverberation, stereo widening, bass boost, etc.) and positional 3D audio that are not part of OpenMAX AL. Similarly, OpenMAX AL has audio features including analog radio tuner and RDS functionality that are not part of OpenSL ES.

The primary focus of OpenSL ES is advanced audio and MIDI functionality for mobile devices. The primary focus of OpenMAX AL is media (audio, video and image) capture and rendering. Further, both OpenSL ES and OpenMAX AL are partitioned into profiles based on market segments:

- OpenSL ES has three overlapping profiles: Phone, Music and Game.
- OpenMAX AL has two overlapping profiles: Media Player and Media Player/Recorder.

Each of these profiles have well-defined feature sets and conformance requirements. For example, to be compliant with the OpenMAX AL Media Player profile, an OpenMAX AL implementation must provide audio, image and video playback functionality. An audio-only OpenMAX AL implementation would not be compliant with either profile of the OpenMAX AL specification.

This segmentation into profiles ensures that there will be no confusion whatsoever regarding which API is suitable for a particular set of use cases.

- Example 1: an audio-only application will have no need for video and image functionality and therefore would likely pick one of the OpenSL ES profiles, depending on the use cases of interest.
- Example 2: a media playback and recording application would use the OpenMAX AL Media Player/Recorder profile.

- Example 3: An advanced multimedia/game application that needs audio/video/image playback and recording as well as advanced audio features such as 3D audio and effects would use both the Media Player/Recorder profile of OpenMAX AL and the Game profile of OpenSL ES.

The two APIs have been designed such that their architecture are identical. Further, each API has identical API methods for the same functionality. At the same time, the APIs are also independent – each can be used as a standalone API by itself (as in Examples 1 and 2) or can co-exist with the other on the same device (as in Example 3).

1.6 Conventions Used

When this specification discusses requirements and features of the OpenSL ES API, specific words are used to indicate the requirement of a feature in an implementation. The table below shows a list of these words.**Requirement Terminology**

Word	**Definition**
May	The stated functionality is an optional requirement for an implementation of the OpenSL ES API. Optional features are not required by the specification but may have conformance requirements if they are implemented. This is an optional feature as in “The implementation *may* have vendor-specific extensions.”
Shall	The stated functionality is a requirement for an implementation of the OpenSL ES API. If an implementation fails to meet a *shall* statement, it is not considered as conforming to this specification. *Shall* is always used as a requirement, as in “The implementation *shall* support the play interface.”
Should	The stated functionality is not a requirement for an implementation of the OpenSL ES API but is recommended or is a good practice. *Should* is usually used as follows: “An OpenSL ES implementation of the game profile *should* be capable of playing content encoded with an MP3 codec.” While this is a recommended practice, an implementation could still be considered as conforming to the OpenSL ES API without implementing this functionality.
Will	The stated functionality is not a requirement for an implementation of the OpenSL ES API. *Will* is typically used when referring to a third party, as in “the application framework *will* correctly handle errors.”
Deprecated	Some APIs are marked as "deprecated". This means that the API was defined in an earlier version of the specification, but is no longer recommended for use. In most cases, a preferred alternative API is available. Implementations are permitted to continue providing the deprecated API, and applications are encouraged to convert from the deprecated API to the recommended alternative API. Deprecated APIs are likely to be removed completely in a subsequent version.

1.6.1 Parameter Range Notation

Valid parameter ranges are specified using both enumerated lists of valid values and sequential ranges. The ranges are specified using the following interval notation [ISO31-11]: (a, b) for open intervals, [a, b] for closed intervals, and (a, b] and [a, b) for half-closed intervals, defined as follows:

$$(a,b) = \{x \mid a < x < b\}$$
$$[a,b] = \{x \mid a \le x \le b\}$$
$$(a,b] = \{x \mid a < x \le b\}$$
$$[a,b) = \{x \mid a \le x < b\}$$

1.6.2 Format and Typographic Conventions

This document uses the following conventions:

Table 2: Format and Typographic Conventions

Format	Meaning
`Courier font`	Sample code, API parameter specifications

1.7 Definition of Terms

The following terms in are used in this specification:

Table 3: Specification Terminology

Word	Definition
3D source	A player or group of players positioned in 3D space.
Application	A software program that makes calls to the OpenSL ES API. An application is the client of the API.
Implementation	A realization of the OpenSL ES specification. One example is a software library that conforms to the OpenSL ES specification.
Platform	A software and hardware architecture that supports running applications.

1.8 Acknowledgements

The OpenSL ES specification is the result of the contributions of many people. The following is a partial list of contributors, including the respective companies represented at the time of their contribution:

Erik Noreke, Independent (Chair)
Stewart Chao, AMD (now with Qualcomm)
Tom Longo, AMD (now with Qualcomm)
Wilson Kwan, AMD (now with Qualcomm)
Chris Grigg, Beatnik
Andrew Ezekiel Rostaing, Beatnik
Roger Nixon, Broadcom
Tim Granger, Broadcom
Wolfgang Schildbach, Coding Technologies
Peter Clare, Creative
Robert Alm, Ericsson
Lewis Balfour, Ericsson
Harald Gustafsson, Ericsson
Håkan Gårdrup, Ericsson
Jacob Ström, Ericsson
Pierre Bossart, Freescale
Brian Murray, Freescale
Glenn Kasten, Google
Jean-Michel Trivi, Google
Jarmo Hiipakka, Nokia
Yeshwant Muthusamy, Nokia
Matti Paavola, Nokia
Scott Peterson, NVIDIA
Neil Trevett, NVIDIA
Jim Van Welzen, NVIDIA
Gregor Brandt, QSound
Brian Schmidt, QSound
John Mortimer, QSound
Mark Williams, QSound
Scott Willing, QSound
Ytai Ben-Tsvi, Samsung (Past Editor)
Natan Linder, Samsung
Gal Sivan, Samsung
Weijun Wang, SKY MobileMedia
Tim Jones, Sonaptic
Jonathan Page, Sonaptic
Stephan Tassart, ST Microelectronics
Tim Renouf, Tao Group
David Eaton, SRS Labs
Jefferson Hobbs, SRS Labs (Editor)
Steven Winston, SRS Labs (Past Editor)
Brian Evans, Symbian
Pavan Kumar, Symbian

James Ingram, Symbian
Rajeev Ragunathan, Symbian
Leo Estevez, Texas Instruments
Danny Jochelson, Texas Instruments
Howard Yeh, Qualcomm
Nathan Charles, ZiiLABS (Past Chair)

2 Features and Profiles

OpenSL ES was designed with audio application developers in mind. It provides support for a number of audio features that facilitate the development of a wide range of applications on the target devices. Supported features include:

- **Playback of audio and MIDI:** Includes playback of PCM and encoded content, MIDI ringtones and UI sounds, as well as extraction of metadata embedded in the content.
- **Effects and controls:** Includes general audio controls such as volume, rate, and pitch, music player effects such as equalizer, bass boost, preset reverberation and stereo widening, as well as advanced 3D effects such as Doppler, environmental reverberation, and virtualization.
- **Advanced MIDI:** Includes support for SP-MIDI, mobile DLS, mobile XMF, MIDI messages, and the ability to use the output of the MIDI engine as a 3D sound source.
- **3D Audio:** Includes 3D positional audio and 3D groups (grouping of 3D sound sources). 3D audio functionality facilitates the use of OpenSL ES as the audio companion to OpenGL ES (a 2D/3D graphics API from Khronos) for gaming.

OpenSL ES also provides optional support for LED and vibra control, 3D macroscopic control, and audio recording. Audio recording - in PCM as well as non-PCM formats – can be from a microphone, a line-in jack. Section 2.4 discusses optional features in the API.

These audio features enable the development of applications such as music players, ring-tone players, voice recorders, simple 2D games as well as advanced 3D games, and MIDI sequencers, to name a few.

2.1 Motivation

The need for segmenting the set of OpenSL ES features into profiles arose from the following considerations:

- The realization that the set of OpenSL ES features was quite large.
- The fact that OpenSL ES would most likely be used for a range of devices catering to quite different market segments. Therefore, not all implementations of OpenSL ES would need (or could accommodate) all of the functionality represented by this large set of features.

Segmentation of the API into profiles based on market segments is considered a better approach than one based solely on technical criteria.

2.2 Profile Definition

An OpenSL ES profile is a defined subset of features that satisfy typical use cases for a given market segment. Any feature may be included in any profile, and any device may support any number of profiles.

2.3 Profiles

OpenSL ES is segmented into three profiles: **Phone, Music** and **Game**. A short description and rationale of each of the profiles is discussed below:

- **Phone:** This is the basic profile that is designed for the low-end or "basic" mobile phone market segment. This includes ringtone and alert tone playback (that is, basic MIDI functionality), basic audio playback, and simple 2D audio games. Recording functionality, which is commonly used for recording voice memos on mobile phones, is an optional feature and is not part of any profile. Optional features and their relationship to profiles are described in more detail in section 2.4, below.
- **Music:** This profile is designed for the music-centric mobile device market. Characteristics of such devices include high-quality audio, the ability to support multiple music audio codecs, and the ability to play music from local files. Some high-end devices could also support streaming audio from remote servers (although this is not mandated functionality in OpenSL ES). A mobile phone that has a built-in music player would incorporate both the Phone and Music profiles. A digital music-only mobile device would use only the Music profile.
- **Game:** This profile is designed for the game-centric mobile device market. Characteristics of such devices include advanced MIDI functionality, and sophisticated audio capabilities such as 3D audio, audio effects, and the ability to handle buffers of audio. A mobile phone that offers sophisticated game-playing ability would incorporate both the Phone and Game profiles. A game-only device would use only the Game profile.

Other combinations of these three profiles are also possible -- a full-featured game-and-music mobile phone would incorporate all three profiles: Phone, Music and Game, and a PDA handheld with only wireless network capability but no mobile phone functionality might incorporate only the Music and Game profiles.

It is worth noting here that these three profiles are by no means distinct (and thus should not be confused with levels); they do indeed overlap – mostly in basic audio API functionality, but also in areas that reflect the increasing sophistication in even the "basic" mobile phones that are likely to be in the marketplace for the near future. Conversely, none of the profiles is a superset of one of the other two profiles.

The following table lists the features in the three profiles of OpenSL ES. A "Y" in a cell indicates that the listed API feature is mandatory in that profile, while a blank cell indicates the absence of that feature.

Table 4: Features of the Three OpenSL ES Profiles

API (Profile) Feature	Phone	Music	Game
PLAYBACK AND PROCESSING CONTROLS			
Play multiple sounds at a given time	Y	Y	Y
Playback of mono and stereo sounds	Y	Y	Y
Basic playback controls	Y	Y	Y
End to end looping	Y	Y	Y
Segment looping			Y
Query and set the playback position	Y	Y	Y
Position-related callbacks and notifications	Y	Y	Y
Sound prioritization	Y	Y	Y
Route audio to multiple simultaneous outputs	Y		
Volume control	Y	Y	Y
Audio balance and pan control		Y	Y
Metadata retrieval		Y	Y
Modify playback pitch and rate			Y
Playback of sounds in secondary storage	Y	Y	Y
Buffers & buffer queues			Y
CAPABILITY QUERIES			
Query capabilities of the implementation	Y	Y	Y
Enumerate audio I/O devices	Y	Y	Y
Query audio I/O device capabilities	Y	Y	Y
EFFECTS			
Stereo widening		Y	Y
Virtualization		Y	Y
Reverberation		Y	Y
Equalization		Y	Y
Effect send control		Y	Y
MIDI			
Support for: SP-MIDI, Mobile DLS, Mobile XMF	Y		Y

API (Profile) Feature	Phone	Music	Game
MIDI messages	Y		Y
MIDI tempo	Y		Y
MIDI buffer queues	Y		Y
Adjustable MIDI tick length	Y		Y
JSR-135 tone control	Y		Y
MIDI track and channel mute/solo			Y
3D AUDIO			
Positional 3D audio			Y
Sound cones			Y
Multiple distance models			Y
Source and listener velocity			Y
Source and listener orientation			Y
3D sound grouping			Y
Simultaneous commit of multiple 3D controls			Y

2.4 Optionality Rules of Features and Profiles

To minimize confusion among developers and reduce fragmentation, the API adheres to the following rules on features and profiles:

1. All features within a profile are mandatory – this is critical in assuring developers and implementers that if they pick a profile, all the functionality representative of that profile will indeed be present. Then, applications written to specific profile(s) will indeed be portable across OpenSL ES implementations of those profile(s).
2. A feature that does not fit in any of the profiles is considered optional in all profiles of OpenSL ES. It is also best not to further categorize optional features in any way, to avoid a potentially confusing combinatorial explosion ("Profile X with Optional Feature Categories 1 and 2", or "Profile X with Optional Feature Categories 2 and 3," and so on) and effectively negating the benefits of Rule #1. Vendors are free to pick and choose from the entire set of optional features to augment their implementations of any of the profiles.
3. Vendors are free to implement features from more than one profile, but they can claim compliance with a profile only if they implement all of the features of that profile. Example: if a vendor implements the Phone profile in its entirety but just three of the features in the Music profile, then that vendor can claim compliance only with the Phone profile.

The following table lists some of the optional features in OpenSL ES 1.1 with the reason for their optionality.

Table 5: Optional Features in OpenSL ES

Optional Feature	Reason for Optionality
Audio recording from a microphone or on-device, line-in jack	Implies hardware dependency (presence of a microphone or line-in jack)
Haptics – support non-audio output devices such as LEDs and vibrator(s)	Implies hardware dependency (presence of LEDs and vibrator(s))
Macroscopic behavior in 3D audio	Cutting-edge feature for mobile devices that will be supported by advanced implementations

2.5 Profile Notes

Profiles notes are used within this specification to identify objects and interfaces where support is optional in one or more of the profiles. Profile notes are not used where an object or interface is mandated in all three profiles. Here are some representative examples of profile notes found in the specification:

PROFILE NOTES
This object is a standardized extension and consequently optional in all profiles.

PROFILE NOTES
This interface is mandated only in the Music and Game profiles.

2.6 Behavior for Unsupported Features

If an application attempts to use a feature that is not present in a specific implementation of OpenSL ES, the implementation shall fail the request to use the feature, returning the `SL_RESULT_FEATURE_UNSUPPORTED` error code [see section 3.4 on Error Reporting]. This can happen either when calling `GetInterface()` on an unsupported interface or when attempting to call a method not supported in an interface. Further, if an interface with an unknown ID is used (either during object creation or in a `GetInterface()` call), this same result code shall be returned. This facilitates portability of applications using non-standard extensions.

3 Design Overview

3.1 Object Model

3.1.1 Objects and Interfaces

The OpenSL ES API adopts an object-oriented approach using the C programming language. The API includes two fundamental concepts on which are built all other constructs: an object and an interface.

An **object** is an abstraction of a set of resources, assigned for a well-defined set of tasks, and the state of these resources. An object has a type determined on its creation. The object type determines the set of tasks that the object can perform. This can be considered similar to a class in C++.

An **interface** is an abstraction of a set of related features that a certain object provides. An interface includes a set of **methods**, which are functions of the interface. An interface also has a type which determines the exact set of methods of the interface. We can define the interface itself as a combination of its type and the object to which it is related.An **interface ID** identifies an interface type. This identifier is used within the source code to refer to the interface type.

Objects and interfaces are tightly related – an object **exposes** one or more interfaces, all of which have different interface types -- that is, an object may contain at most one interface of each interface type. A given interface instance is **exposed** by exactly one object. The application controls the object's state and performs the object operations exclusively through the interfaces it exposes. Thus, the object itself is a completely abstract notion, and has no actual representation in code, yet it is very important to understand that behind every interface stands an object.

The relationship between object types and interface types is as follows. The object type determines the types of interfaces that may be exposed by objects of this type. Each object type definition in this document includes a detailed list of the interfaces on that object.

PROFILE NOTES
The set of interface types allowed for a given object type may vary across profiles. This will be explicitly stated in this document, per object, and per interface type.

An object's **lifetime** is the time between the object's **creation** and its **destruction**. The application explicitly performs both object creation and destruction, as will be explained later in this document.

An object maintains a state machine with the following states:

- **Unrealized (initial state):** The object is alive but has not allocated resources. It is not yet usable; its interfaces' methods cannot yet be called.
- **Realized:** The object's resources are allocated and the object is usable.

- **Suspended (optional state):** The object has fewer resources than it requires in order to be usable but it maintains the state it was in at the moment of suspension. This state is optional to the extent that, in the face of resource loss, the implementation has the option of putting an object either in the Suspended state or the Unrealized state.

The following state diagram illustrates the states and transitions.

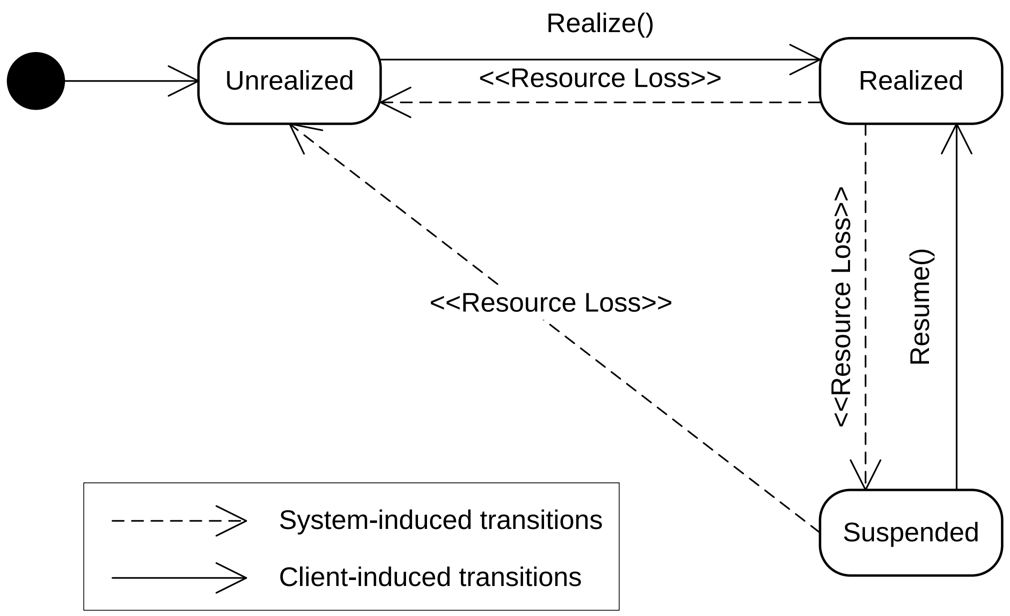


Figure 2: Object state diagram

When the application destroys an object, that object implicitly transitions through the Unrealized state. Thus it frees its resources and makes them available to other objects.

See section 3.1.7 for more details on resource allocation.

The following example demonstrates the transition between object states:

```
SLresult        res;
SLAudioPlayer   player;
SLPlayItf       playbackItf;
SLint32         priority;
SLboolean       preemptable;
SLuint32        state;
SLmillibel      vol;

/* create an audio player */
res = (*eng)->CreateAudioPlayer(eng, ..., &player);
CheckErr(res);
/* Realizing the object in synchronous mode. */
res = (*player)->Realize(player, SL_BOOLEAN_FALSE);
...
```

```
...
...
...
/* Checking the object's state since we would like to use it now, and its
resources may have been stolen. */
res = (*player)->GetState(player, &state);
if (SL_OBJECT_STATE_SUSPENDED == state)
{
   /* Resuming state synchronously. */
   res = (*player)->Resume(player, SL_BOOLEAN_FALSE);
   while (SL_RESULT_RESOURCE_ERROR == res)
   {
         /* Not enough resources. Increasing object priority. */
         res = (*player)->GetPriority(player, &priority, &preemptable);
         res = (*player)->SetPriority(player, INT_MAX, SL_BOOLEAN_FALSE);
         /* trying again */
         res = (*player)->Resume(player, SL_BOOLEAN_FALSE);
   }
} else
{
   if (SL_OBJECT_STATE_UNREALIZED == res)
   {
         /* Realizing state synchronously. */
         res = (*player)->Realize(player, SL_BOOLEAN_FALSE);
         while (SL_RESULT_RESOURCE_ERROR == res)
         {
               /* Not enough resources. Increasing object priority. */
               res = (*player)->GetPriority(player, &priority,
&preemptable);
               res = (*player)->SetPriority(player, INT_MAX,
SL_BOOLEAN_FALSE);
               /* trying again */
               res = (*player)->Realize(player, SL_BOOLEAN_FALSE);
         }
   }
}
```

3.1.2 Getters and Setters

Getters and setters provide access to object properties. An application uses a setter to change the value of an object's property and a getter to retrieve a value of an object's property.

Unless explicitly specified in a getter's method name, a getter shall always return the exact value that was set by a previous call to a setter, even if that value had been rounded off internally by the implementation. An exception to this rule is that a Boolean getter must return only logically (but not necessarily numerically) the same value as was set.

Here is a short example that demonstrates the use of a getter and a setter:

```
SLresult        res;
SLVolumeItf     volumeItf;
SLmillibel      vol;

res = (*volumeItf)->GetVolumeLevel(volumeItf, &vol); CheckErr(res);
res = (*volumeItf)->SetVolumeLevel(volumeItf, vol + 7); CheckErr(res);
```

Unless specified otherwise, applications are responsible for the allocation and deallocation of memory buffers used in the interface methods.

3.1.3 Representation in Code

As stated in the previous section, objects have no representation in code. OpenSL ES refers to an object via its `SLObjectItf` interface [see sections 3.1.4 and 8.34].

The API represents interfaces as C-structs, where all the fields are method-pointers, representing the methods. These interface structures are always stored and passed as pointer-to-pointer-to-struct and never by value (this level of indirection enables more efficient API implementations).

Each of the interface methods has a first argument, called `self`, whose type is the interface type. Thus when calling an interface method, the caller must pass the interface pointer in this argument. Additionally, each of the callback prototypes has a first argument called `caller`, whose type is the interface type. Here is an example of a simple interface:

```
struct SLSomeInterfaceItf_;
typedef const struct SLSomeInterfaceItf_ * const * SLSomeInterfaceItf;

struct SLSomeInterfaceItf_ {
   SLresult (*Method1) (
         SLSomeInterfaceItf self,
         SLint32 prm
   );
   SLresult (*Method2) (
         SLSomeInterfaceItf self
   );
};
```

This interface is called `SLSomeInterfaceItf` and has two methods: `Method1()` and `Method2()`. Such an interface will be used as follows:

```
SLuint32 i;
SLresult res;
SLSomeInterfaceItf someItf;

/* ... obtain this interface somehow ... */
res = (*someItf)->Method1(someItf, 13);
res = (*someItf)->Method2(someItf);
```

Note that this code excludes the mechanism for obtaining the interface itself, which will be explained in the following sections.

3.1.4 The SLObjectItf Interface

`SLObjectItf` is a special interface type fundamental to the architecture. Every object type in the API **exposes this interface**. It is the "entry-point" for an object since every method that **creates** a new object actually returns the `SLObjectItf` of this object. Using this interface, an application may perform all the basic operations on the object and may access other interfaces exposed by the object.

The application **destroys** the object by calling the `Destroy()` method of the `SLObjectItf` interface.

The application **obtains other interfaces** of the object by calling the `GetInterface()` method of the `SLObjectItf` interface. The application retrieves an interface by its type ID which uniquely identifies it; an object cannot contain more than one interface of a certain type.

The application **controls the state** of the object by calling the `Realize()` and `Resume()` methods of the `SLObjectItf` interface.

For a complete and detailed specification of this interface type, see section 8.34.

3.1.5 The Engine Object and SLEngineItf Interface

Other fundamental entities in the architecture are the engine object and the `SLEngineItf` interface. These entities serve as the API's entry-point. The application begins an OpenSL ES session by creating an engine object.

The engine object is created using the global function `slCreateEngine()` [see section 6.1]. The result of engine object creation is a `SLObjectItf` interface, regardless of the implementation.

An implementation can optionally support multiple versions of the API. When the application creates the engine object, it specifies the version of the engine object. If the requested version of the API is supported, an engine object of that version is returned. Any object or interface created or derived from that engine object will adhere to the same version of the specification as the engine object.

After creating the engine object, the application can obtain this object's `SLEngineItf` interface. This interface contains creation methods for all the other object types in the API.

To create an object process:

- Create and realize an engine object if one has not been created already.
- Obtain the `SLEngineItf` interface from this object.

- Call the appropriate object creation method of this interface.
- If the call succeeds, the method will return an `SLObjectItf` interface of the new object.
- After working with the object, call the `Destroy()` method of the `SLObjectItf` to free the object and its resources.

For a complete and detailed specification of the engine object and the `SLEngineItf` interface type, please refer to sections 7.4 and 8.21 respectively.

3.1.6 The Relationship Between Objects and Interfaces

The set of interfaces exposed by an object is determined by three factors:

- The object's type
- The interfaces requested by the application during the object's creation
- The interfaces added and removed by the application during the object's lifetime

An object's type determines the set of interfaces that will always exist, regardless of whether the application requests them or not. These interfaces are called *implicit interfaces* and every object type has its own set of implicit interfaces that will exist on every object of this type. The `SLObjectItf` interface, introduced in section 3.1.4, is fundamentally required on every object so it is designated as implicit on all object types -- that is, the application never needs to explicitly request that it be exposed on any object.

Every object type also defines a set of interfaces that are available on objects of this type, but will not be exposed by an object unless explicitly requested by the application during the object's creation. These explicitly requested interfaces are called *explicit interfaces*.

Finally, every object type also defines a set of interfaces that may be added and removed by the application during the object's lifetime. These types of interfaces are called *dynamic interfaces* and they are managed by a dedicated interface, called `SLDynamicInterfaceManagementItf` [see section 8.17], which enables this dynamic binding. Attempts to dynamically add or remove implicit interfaces on an object will fail.

The set of explicit and dynamic interfaces for a given object type may vary between implementations [see section 3.6]. However, for a given profile, each object type has a set of mandated explicit interfaces and a set of mandated dynamic interfaces that shall be supported by every implementation.

When an application requests explicit interfaces during object creation, it can flag any interface as required. If an implementation is unable to satisfy the request for an interface that is not flagged as required (i.e. it is not required), this will not cause the object to fail creation. On the other hand, if the interface is flagged as required and the implementation is unable to satisfy the request for the interface, the object will not be created.

The following table summarizes whether an object is created and an interface is exposed, according to how the specification relates their types and how the application designates the interface at the object's creation[1].

[1] The reader is advised that there are known on-screen rendering issues with the following tables at certain screen resolutions and it may be necessary to increase the zoom level to fully view the table cell borders.

Table 6: Interface Exposure Rules During Object Creation

			Determined by the application		
			Interface requested by application		**Interface not requested by application**
			Interface marked as required	**Interface not marked as required**	
Determined by implementation & specification	**Mandated interface**	**Implicit**	✓	✓	✓
		Explicit	✓	✓	✗
	Optional interface	**Available**	✓	✓	✗
		Not available	💣	✗	✗

Key:

✓	Object is created and interface is exposed, subject to resource constraints
✗	Object is created but interface is not exposed
💣	Object is not created and interface is not exposed

The next table summarizes whether interface is exposed on an object when the application requests to add the interface dynamically, according to whether the specification mandates support for the interface on the object type and whether this support is mandated dynamically or not.

Table 7: Interface Exposure Rules for Dynamic Adding of Interface

				Determined by application
				Application dynamically adds interface
ned by impleme ntation & specifica	**Mandated dynamic**	**Mandated interface**		✓
		Optional	**Available**	✓

		interface	**Not available**	✘
	Not mandated dynamic			?

Key:

✓	Interface is exposed, subject to resource constraints
✘	Interface is not exposed
?	Interface may be exposed (implementation dependent), subject to resource constraints

3.1.7 The SLDynamicInterfaceManagementItf Interface

The `SLDynamicInterfaceManagementItf` interface provides methods for handling interface exposure on an object after the creation and realization of the object. The `SLDynamicInterfaceManagementItf` itself is an implicit interface on all object types.

The dynamic nature of an interface is unrelated to it being optional or mandated for an object. An interface that is “mandated” as dynamic can actually be realized both at object creation time as well as dynamically, at any point in the object‘s lifetime (using the `SLDynamicInterfaceManagementItf`). Interfaces that represent a significant resource drain if realized on object creation but that are never used, are prime candidates for dynamic interfaces. By making them dynamic, the application developer can use them only when needed, often resulting in significant resource optimization. Dynamic interfaces are explicitly called out in the “Mandated Interfaces” sections of the corresponding objects in Section 7.

Although this interface provides a method for requesting the acquisition of an additional interface, namely `AddInterface()`, the implementation may deny the request. The criteria for denial are implementation-dependent. For example, the state of an object’s player or recorder may influence the success or failure of dynamic interface addition. Upon a successful `AddInterface()` call for a specified interface, that interface is immediately usable. There is no separate call to “realize” the given interface. The interface instance is obtained, just as for static interfaces, by using the `GetInterface()` method.

An application may retire a dynamic interface with a `RemoveInterface()` call. After a `RemoveInterface()` call, the dynamic interface is no longer usable. When an object is unrealized, all interfaces (including the dynamic interfaces) are unusable and effectively removed from the object.

3.1.8 Resource Allocation

The exact amount of resources available on an OpenSL ES implementation may vary across different implementations and over the lifetime of an engine object. As a result, an application using the OpenSL ES API must always be prepared to handle cases of failure in object realization or dynamic interface addition. In addition, an object's resources may be stolen by another entity (such as another object, or the underlying system) without prior notice.

To allow applications to influence the resource allocation by the system, a priority mechanism is introduced. Priority values are set on a per-object basis. Applications can use these object priorities to influence the behavior of the system when resource conflicts arise. For example, when a high-priority object needs a certain resource and the resource is currently assigned to a lower-priority object, the resource will most likely be "stolen" from the low-priority object (by the system) and re-assigned to the high-priority object. An application can change the priority of an object at any point during the lifetime of the object. It is also worth noting that these object priorities set by the application are confined to this instance of the API engine. They are unrelated to the priorities that may be assigned by the system to the application itself and other components running on the system, for the purposes of resource management.

When a resource is stolen from an object, this object will automatically transition to either the Suspended state or the Unrealized state, depending on whether its interface states are preserved or reset, respectively. To which of the states the object transitions is determined solely by the implementation. When in either of these two states, all of this object's interfaces, except for the `SLObjectItf` interface and the `SLDynamicInterfaceManagementItf` interface, become unusable, and return an appropriate error code. Dynamic interfaces are treated the same as any other interface type. If the object the dynamic interface is exposed on is suspended or unrealized, the dynamic interfaces will be suspended or unrealized, respectively.

The application may request to be notified of such a transition -- this is done by registering for a notification on the object. The application may also request to be notified when resources become available again, which may allow for the object to regain usability. The notification will include any dynamic interfaces as well, that is, the notification is sent when all the interfaces and the object can have their resources. Individual notification is NOT sent for each dynamic interface.

The application may attempt to recover an Unrealized or Suspended object by calling its `Realize()` or `Resume()` methods, respectively. If the call succeeds, the object will return to the Realized state, and its interface states will be either recovered or reset to default, depending on whether it was unrealized or suspended. The `RemoveInterface()` method is special and can be used in any object state to retire dynamically exposed interfaces. This may help in successfully realizing or resuming the object.

When a stolen resource is freed, the implementation checks whether this resource can be used in order to recover an interface in a resources stolen state. The check is made in object priority order, from high to low. It is not guaranteed, however, that attempting to recover an object after getting this notification will succeed.

An important difference regarding interfaces that are exposed dynamically is how resources are managed. When a dynamic interface loses its resources, a notification is sent but the object state is not affected. Also, other interfaces on the same object are not affected. The application may register for notification of dynamic interface resource changes.

After a lost resources notification, the dynamically exposed interface will become unusable. Two different types of lost resources notification can be received– resource lost, and resource lost permanently. The first type of notification indicates that the dynamic interface may be resumed by the application after a resource available notification has been received. When the `ResumeInterface()` call succeeds, the dynamic interface will be fully recovered. The second type of notification means that the current instance of the exposed dynamic interface can not recover from the resource loss and shall be retired by the application.

3.2 Threading Model

3.2.1 Mode of Operation

The OpenSL ES API is generally synchronous. This means that an API method will return only after its operation is complete, and any state changes caused by the method call will be immediately reflected by subsequent calls.

However, in some specific cases, a synchronous operation is not desirable, due to operations that may take a long time. In such cases, the actual termination of the operation will be signaled by a notification. Any state changes caused by the call are undefined between the time of the call and until the time of notification.

Asynchronous functions will be clearly designated as such in their documentation. Otherwise, a synchronous mode of operation should be assumed.

3.2.2 Thread Safety

The OpenSL ES API may operate in one of two modes, which determine the behavior of the entire API regarding reentrancy:

- **Thread-safe mode:** The application may call the API functions from several contexts concurrently. The entire API will be thread-safe – that is, any combination of the API functions may be invoked concurrently (including invocation of the same method more than once concurrently) by multiple application threads, and are guaranteed to behave as specified.
- **Non-thread-safe mode:** The application needs to take care of synchronization and ensure that at any given time a maximum of one API method is being called. The entire API is not thread-safe – that is, the application needs to ensure that at any given time a maximum of one of the API functions is being executed, or else undefined behavior should be expected.

An implementation shall support one or more of these modes.

The mode of operation is determined on engine creation, and cannot be changed during the lifetime of the engine object. An implementation shall support at least one of these modes, and should document which modes are supported.

Note that a application written to work with non-thread-safe mode will be able to work with a thread-safe mode engine without change. As a result, a valid implementation of thread-safe mode is automatically considered a valid implementation of the non-thread-safe mode; however, implementations of both modes may choose to implement them differently. Implementers should note that implementation of thread-safe mode assumes knowledge of the threading mechanisms used by the application.

3.3 Notifications

In several cases, the application needs to be notified when some event occurred inside the OpenSL ES implementation, such as when playback of a file has ended, or when an asynchronous method has completed. These notifications are implemented as callback functions – the application registers a method whose signature is specified by the API and this method will be called by the OpenSL ES implementation every time a certain event occurs.

Callback functions are registered per-interface and per-event type -- thus registering a callback for a certain event on a given object (through one of its interfaces) will not cause this callback to be called if the same event occurs on a different object, or if a different event occurs on the same object. The event type is simply designated by the method that was used to register the callback.

At any given time, a maximum of one callback function may be registered per-interface, per-event type. Registering a new callback on the same interface, using the same registration method, will un-register the old callback. Similarly, registering `NULL` is the way to un-register an existing callback without registering a new one.

The context from which the callbacks are invoked is undefined, and typically implementation- and OS-dependent. Thus the application cannot rely on any system call or OpenSL ES API call to work from within this call. However, to avoid a dead-end, each implementation should document the list of functions that can be safely called from the callback context. It is highly recommended that the implementation provide at least the means of posting messages to other application threads, where the event shall be handled. In addition, the `SLThreadSyncItf` interface [see section 8.44] must be usable from within the callback context.

The application should be aware of the fact that callbacks may be invoked concurrently with other callbacks, concurrently with application invocation of an API method, or even from within API calls, and thus should be prepared to handle the required synchronization, typically using the `SLThreadSyncItf` interface [see section 8.44].

For more specific details, refer to the engine object documentation [see section 7.4].

3.4 Error Reporting

Almost every API method indicates its success or failure by a result code (except for methods that are not allowed to fail under any circumstances). An API method's documentation states the valid result codes for that method and an implementation shall return one of these result codes. For synchronous methods, the result code is the return value of the method. For asynchronous functions, the result code is contained in the data of the notification sent upon the completion of the operation.

Every API method has a set of pre-conditions associated with it, consisting of:

- API state in which the method should be called
- Context from which the method can be called

The pre-conditions are clearly documented for every method. When the application violates any of the pre-conditions, the method call will fail, and the method's result code will indicate the violation. The API will remain stable and its state will not be affected by the call. However, it is recommended that applications do not rely on this behavior and avoid violating the pre-conditions. The main value of this behavior is to aid in the debug process of applications, and to guarantee stability in extreme conditions, and specifically under race-conditions.

However, the API's behavior may be undefined (and even unstable) in any of the following conditions:

- Corruption of the `self` parameter, which is passed as every method's first parameter, or any other parameter passed by pointer.
- Violation of the threading policy.

3.5 Memory Management

3.5.1 General

The application is responsible for allocating and deallocating all memory originating within the application space. OpenSL ES is responsible for allocating and deallocating all memory originating within the OpenSL ES implementation. At no point shall the application deallocate memory allocated by the OpenSL ES implementation, and the OpenSL ES implementation will not deallocate memory allocated by the application. Exceptions to this rule are not allowed.

3.5.2 Parameters

Parameters passed to OpenSL ES methods are used during the duration of the method call. The application is free to change or deallocate any parameters or parameter associated memory after a call to OpenSL ES, with the exception of memory buffers. Memory buffers passed from the application to the OpenSL ES implementation are to be kept valid by the application for the duration of the use by the OpenSL ES implementation. The duration of the memory buffer use and when the application can deallocate the memory is described in the interface or object description where they are used.

3.5.3 Callbacks

The information passed to the application from the OpenSL ES implementation in a callback is valid for the duration of the callback only. The OpenSL ES implementation is free to deallocate any memory

associated with a callback to the application as soon as the callback returns. The application is responsible for saving any information passed to it in a callback required for later use. This includes any memory buffers passed from the OpenSL ES implementation to the application.

3.5.4 Exceptions

Any exceptions and behavior in contrast to the memory management rules stated above will be described in the interface description which behaves in contrast to the above rules.

3.6 Extensibility

3.6.1 Principles

The OpenSL ES API was designed with extensibility in mind. An extended API is defined as one that provides functionality additional to that defined by the specification, yet considered still conforming to the specification.

The main principles of the extensibility mechanism are:

- Any application written to work with the standard API will still work, unchanged, on the extended API.
- For an application that makes use of extensions, it will be possible and simple to identify cases where these extensions are not supported, and thus to degrade its functionality gracefully.

Possible extensions may include vendor-specific extensions as well as future versions of OpenSL ES.

3.6.2 Permitted Modifications to Physical Code

The OpenSL ES header files shall be edited only for the following purpose:

- To amend definitions of types (for example, 32 bit signed integers) such that they have correct representation.

Any vendor-specific extensions to the API shall reside in header files other than the OpenSL ES header files.

3.6.3 Extending Supported Interface Types

An extended API may introduce new interface types and expose these interfaces on either existing object types or on extended object types [see section 3.6.4].

An extended API may also expose standard interfaces on standard / extended object types that do not normally require exposing these interfaces.

The extended interfaces will be defined in a manner similar to standard interfaces. The extended interface types will have unique IDs, generated by the extension provider.

Note that the extended API may not alter standard interfaces or apply different semantics on standard interfaces, even if the syntax is preserved. An exception to this rule is extending the valid parameter range of functions, detailed later.

Functions may not be added to any of the interfaces defined in the specification. To do that, a new interface which includes the desired standard interface must be defined, along with a new interface ID which must be generated.

It is also highly recommended that whenever an interface's signature changes (even slightly), a new interface ID will be generated, and the modified interface will be considered a new one. This is to protect application applications already written to work with the original interface.

3.6.4 Extending Supported Object Types

An extended API may introduce new object types to those specified in the standard API. The extended objects may expose either standard or extended interface types. Should it expose standard interfaces – they must still behave as specified. Otherwise, the extended API may provide extended interface types with different semantics.

The extended objects will be created by utilizing the `CreateExtensionObject()` function in the `SLEngineItf` on the engine object or extended interfaces with creation functions. These extended interfaces typically will be exposed by the standard engine object, but can also be exposed on other objects.

3.6.5 Extending Method Parameter Ranges

An extended API may support a greater range of parameters for a standard method than the range mandated by the specification. The semantics of the extended range are left to the extended API's specification. However, for mandated ranges, the API shall behave exactly according to the specification.

Care must be taken when the extended API is vendor-specific in these cases – future versions of the API may use these extended values for different purposes. To help guard against collisions with future API versions, implementations of an extended API shall have the most significant bit set on any extensions to an enumeration (a fixed set of discrete unsigned values). For example:

```
#define SL_SEEKMODE_FAST         ((SLuint16) 0x0001)
#define SL_SEEKMODE_ACCURATE     ((SLuint16) 0x0002)
/* ACME extension to SEEKMODE enumeration */
#define SL_SEEKMODE_ACME_FOO     ((SLuint16) 0x8001)
```

The most significant bit does not need to be set for any extensions to continuous ranges or for signed values.

3.6.6 Result Codes

It is not possible to extend the result codes for any standardized method in the API. An implementation shall return one of the result codes listed in the method's documentation.

3.6.7 Data Locators

An extended API may introduce new data locators to those specified in the standard API. The extended data locators may be used in place of the data locators defined in the main specification. The extension will specify which objects accept the data locators defined by the extension.

3.6.8 Interface ID Allocation Scheme

A common interface ID allocation scheme shall be used for vendor-specific interface IDs, to prevent collisions by different vendors.

The UUID mechanism provided freely in the Web-site below is highly recommended to be used by all providers of extensions.

http://www.itu.int/ITU-T/asn1/uuid.html

The interface IDs do not have to be registered – it is assumed that the above mechanism will never generate the same ID twice.

3.6.9 Avoiding Naming Collisions

It is recommended that vendors providing extended APIs qualify all the global identifiers and macros they provide with some unique prefix, such as the vendor name. This prefix will come after the API prefix, such as `SLAcmeDistortionItf`.

This is meant to reduce the risk of collisions between vendor-specific identifiers and other versions of the specification of other vendors.

The example on the next page demonstrates using extensible features of the specification. The code will compile both on implementations which support the extended API as well as those which do not.

```
void ShowExtensibility(SLEngineItf *eng)
{
   SLresult                res;
   SLboolean               supported;
   SLAudioPlayer           player;
   SLAcmeDistortionItf     distortionItf;
   SLPlayItf               playbackItf;
   SLmillibel              vol;

   /* create an audio player */
   res = eng->CreateAudioPlayer(eng, ..., (void *)&player);
   CheckErr(res);
   res = (*player)->GetInterface(player, &SL_IID_ACME_DISTORTION,
                                 (void *)&distortionItf);
   if (SL_RESULT_FEATURE_UNSUPPORTED == res)
   {
         supported = false;
   } else
   {
         CheckErr(res);
         supported = true;
   }

   /* continue using the player normally whether the extension
      is supported or not. */
   res = (*player)->GetInterface(player, &SL_IID_PLAYBACK,
                              (void *)&playbackItf);
   CheckErr(res);

   ...
   ...
   ...

   /* whenever calling an extension‘s method,
      wrap it with a condition. */
   if (supported)
   {
         /* employ one of the interface’s methods */
         res = (*distortionItf)->SetDistortionGain(distortionItf, vol);
         CheckErr(res);
   }
}
```

4 Functional Overview

4.1 Object Overview

OpenSL ES represents entities in its architecture as objects, including:

- Engine Object
- Media Objects
- Metadata Extractor Objects
- Audio Output Mix Objects
- LED Array Objects
- Vibration Control Objects

The following sections provide an overview of each of these object types.

4.1.1 Engine Object

The engine object is the entry point to the OpenSL ES API. This object enables you to create all the other objects used in OpenSL ES.

The engine object is special in the sense that it is created using a global function, `slCreateEngine()` [see section 6.1]. The result of the creation process is the `SLObjectItf` interface [see section 8.34] of the engine object [see section 7.4]. The implementation is not required to support the creation of more than one engine at a given time.

The engine object can have two different modes of operation, thread-safe mode and non-thread safe mode. The application specifies the mode of operation upon engine object creation. See section 3.2 for details.

The engine object shall expose the `SLThreadSyncItf` interface [see section 8.44] to enable synchronization between the API's callback contexts and the application contexts.

After creation of the engine object, most of the work will be done with the `SLEngineItf` interface [see section 8.21] exposed by this object.

An additional functionality of the engine object is querying implementation-specific capabilities. This includes the encoder and decoder capabilities of the system. The OpenSL ES API gives implementations some freedom regarding their capabilities, and these capabilities may even change according to runtime conditions. For this reason, during runtime the application may query the actual capabilities. However, this specification defines a minimum set of capabilities, expressed as a set of use-cases that shall be supported on every implementation, according to the profiles that are implemented. These use-cases are described in detail in section 4.7.

4.1.1.1 Devices

The engine object represents the system's various multimedia-related devices via unique device IDs. It supports the enumeration of audio input, audio output, LED and vibrator devices as well as mechanisms to query their capabilities. Applications can use information regarding the device capabilities to:

- Determine if they can even run on the system (for example, an application that can render only 8 kHz 8-bit audio might not be able to run on a system that can handle only sampling rates of 16 kHz and above at its outputs.)
- Configure the user interface accordingly so that the user is presented with the correct device choices in the UI menus.

The audio I/O device capabilities interface is described in section 8.12. The LED and Vibra IO device capabilities are described in section 8.22.

4.1.2 Media Objects

A media object implements a multimedia use case by performing some media processing task given a prescribed set of inputs and outputs. Media objects include (but are not limited to) objects that present and capture media streams, often referred to as *players* and *recorders*, respectively. They operate on audio data.

The following characteristics define a media object:

- The operation it performs, denoted by the creation method used to instantiate the media object.
- The inputs it draws data from, denoted as its *data sources* and specified at media object creation.
- The outputs it sends data to, denoted as its *data sinks* and specified at media object creation.

The media object creation methods are described in section 8.21. The audio player object is documented in section 7.2, the MIDI player object is documented in section 7.8 and the audio recorder object is documented in section 7.3.

4.1.2.1 Data Source and Sink Structures

A data source is an input parameter to a media object specifying from where the media object will receive a particular type of data (such as sampled audio or MIDI data). A data sink is an input parameter to a media object specifying to where the media object will send a particular type of data.

The number and types of data sources and sinks differ from one media object to another. The following characteristics define a data source or sink:

- Its *data locator* which identifies where the data resides. Possible locators include:
  - URIs (such as a filename)
  - Memory addresses
  - I/O devices
  - Output Mixes
  - Buffer queues
  - Content pipes
  - NULL
- Its *data format* which identifies the characteristics of the data stream. Possible formats include:
  - MIME-type based formats
  - PCM formats

An application specifies a media object's respective data source(s) and sink(s) upfront in the creation method for the media object. Collectively, the media object together with its associated source(s) and sinks(s) define the use case the application wants executed.

Piped content is supported for data sources and data sinks through the use of a content pipe data locator, as defined in the Content Pipe Specification [CP]. Support for content pipe data locator is an optional feature for all profiles.

Data sources are documented in section 9.1.19 and data sinks are documented in section 9.1.18.

4.1.3 Metadata Extractor Object

Player objects support reading of the metadata of the media content. However, sometimes it is useful just to be able to read metadata without having to be able to playback the media. A Metadata Extractor object can be used for reading metadata without allocating resources for media playback. Using this object is recommended particularly when the application is interested only in presenting metadata without playing the content and when wanting to present metadata of multiple files. The latter is useful for generating playlists for presentation purposes because a player object would unnecessarily allocate playback resources. Furthermore, the ability to change a data source dynamically is an optional feature, which when not available will result in the creation and destruction of many individual player objects in order to extract metadata from multiple files which is inefficient. A Metadata Extractor object does not have a data sink, but it has one data source that can be dynamically changed.

The metadata extractor object is documented in section 7.7 and the metadata extraction interfaces are described in sections 8.26.

4.1.4 Audio Output Mix Object

The API allows for routing of audio to multiple audio outputs and includes an audio output mix object that facilitates this functionality. The application retrieves an output mix object from the engine and may specify that output mix as the sink for a media object. The audio output mix object is specified as a sink for a media object using the `SL_DATALOCATOR_OUTPUTMIX` data locator as described in section 9.2.12. The engine populates the output mix with a set of default audio output devices. The application may query for this list of devices or request changes to it via the `SLOutputMixItf` interface. The API does not provide a direct audio output IO-device as a sink for media objects.

The audio output mix object is defined in section 7.9 and the output mix interface is described in section 8.35.

4.1.5 LED Array Control Object

Control of the device's LEDs is handled via the LED array object. Querying the capabilities of and creating a LED array object is an engine-level operation, while control over individual LEDs is handled by the object.

The LED array object is documented in section 7.5 and the LED array interface is documented in section 8.25.

4.1.6 Vibration Control Object

Control of the device's vibration support is handled via the Vibra object. Querying the capabilities of and creating a Vibra object is an engine-level operation, while control of the actual vibration is handled by the object.

The vibra object is documented in section 7.10 and the vibra interface is documented in section 8.45.

4.2 Sampled Audio

This section introduces OpenSL ES functionality for the playback and recording of sampled audio content.

An audio player [see section 7.2] is used for sampled audio playback. OpenSL ES supports both file-based and in-memory data sources, as well as buffer queues, for efficient streaming of audio data from memory to the audio system. The API supports data encoded in many formats, although the formats supported by a device are implementation-dependent. An audio player can also be used in order to play back pre-created Java Tone Sequences.

An audio recorder [see section 7.3] is used for capturing audio data. Audio capture is an optional component of OpenSL ES, so some devices may fail creation of audio recording objects.

4.2.1 Recommended Codec

An OpenSL ES implementation of the game profile should be capable of playing content encoded with an MP3 codec (including MPEG-1 layer-3 [MPEG1], MPEG-2 layer-3 [MPEG2], and MPEG-2.5 layer-3 variants). (For a definition of the word “should”, see section 1.5). Please note that MPEG-2.5 layer-3 is not useful for 3D audio rendering since it only supports sampling rates of 12 kHz and below.

4.3 Playback of MIDI

OpenSL ES supports MIDI playback using the standard player creation mechanism, the creation method `SLEngineItf::CreateMIDIPlayer()` [see section 8.21]. This method provides the ability to specify a MIDI data source and an audio output device, as well as an optional data source for an instrument bank data source and data sinks for an LED array output device and a Vibra output device. OpenSL ES supports MIDI data sources that refer to files (SP-MIDI [SP-MIDI] and Mobile XMF [mXMF]) and MIDI buffer queues. Playback properties are controlled via the standard OpenSL ES player interfaces, such as `SLVolumeItf` [see section 8.48], `SLPlayItf` [see section 8.37], `SLPlaybackRateItf` [see section 8.38], and `SLSeekItf` [see section 8.43]. MIDI players also support metadata extraction via the `SLMetadataExtractionItf` [see section 8.26]. The MIDI player object is documented in section 7.8.

4.3.1 Support of Mobile DLS

OpenSL ES supports Mobile DLS [mDLS] soundbanks as stand-alone files provided to a MIDI player on creation or embedded within a Mobile XMF file. In addition, the MIDI player supports the GM soundbanks [MIDI] by default.

In several cases, a MIDI player will not be able to handle two DLS banks at the same time (for example, bank provided during MIDI player creation and bank embedded in the content). In such a case, player creation may fail, and the application can retry the creation without providing the additional bank.

When a program is selected for a MIDI channel (using bank select / program change messages), the MIDI player will first look for the program in the embedded DLS bank, if such exists. If it is not found, the MIDI player will look in the DLS bank that was provided on creation, if applicable. If it is still not found, the MIDI player will try to load the program from the built-in GM bank. If the program does not exist there either, the MIDI player shall generate silence on the specified channel, but should still keep track of that channel's state.

4.3.2 Programmatic Control of the MIDI Player

In addition to file playback, the application can generate MIDI data programmatically and pass it to a MIDI player in one of two methods:

Real-time messages are passed through the `SLMIDIMessageItf` [see section 8.29] interface. Such messages lack any timing information, and are executed by the MIDI player as soon as they are sent. The play state does not affect passing real-time MIDI messages; therefore, it is not necessary to even instantiate SLPlayItf if only real-time messages are used to control the MIDI player (with or without a soundbank source).

Time-stamped messages are passed through buffer-queues. The application must create the MIDI player with a buffer queue source, and then enqueue buffers containing time-stamped MIDI messages. The format of the buffers is identical to the format of the content of a `MTrk` chunk (not including any headers, just the raw time-message-time-message-… data), as defined in the MIDI specification. When working in this manner, the application must manually set the initial tempo and tick resolution if it does not want the defaults, through the `SLMIDITempoItf` interface [see section 8.31].

4.4 3D Audio

Using two ears, a human can determine the direction of a sound source in three-dimensional (3D) space. The acoustic waves that arrive at each ear cause movement of the eardrums, which in turn can be represented as a stereo audio signal that incorporates the 3D information.

OpenSL ES provides the ability to position and move sound sources in 3D space. Implementations use this information so that when the sound sources are rendered, they appear to the listener of the sounds to originate from their specified locations. The exact algorithm used to synthesize the 3D position is implementation-dependent. However, a number of techniques exist, with HRTF-based (**H**ead-**R**elated **T**ransfer **F**unction) synthesis being a common technique.

Different sound sources have different 3D acoustic characteristics. To help model these, OpenSL ES exposes a rich set of tunable 3D parameters. For example, applications can:

- modify a sound source's distance model and its sound cones using the `SL3DSourceItf` interface [see section 8.8];
- adjust the amount of Doppler heard when a sound source passes a listener using the `SL3DDopperItf` interface [see section 8.3];
- set the size of a sound source using the macroscopicity interface `SL3DMacroscopicItf` [see section 8.7].

OpenSL ES is an ideal companion to 3D graphic APIs such as OpenGL ES. The 3D graphics engine will render the 3D graphics scene to a two-dimension display device, and the OpenSL ES implementation will render the 3D audio scene to the audio output device.

4.4.1 3D Sources

4.4.1.1 3D Groups

OpenSL ES supports two methods for controlling the 3D properties of a player. The first method involves directly exposing 3D interfaces on the player object, the 3D properties of which are then controlled using these interfaces.

Sometimes developers will want to position several sound sources with the same 3D properties at the same location. For example, a helicopter may have three sound sources: the engine, main rotor blade, and tail rotor blade. These sound sources are positioned sufficiently close to each other that at a certain distance from the listener all the sound sources can be approximated to one location. The application could use three separate players but this is likely to result in the use of three separate, potentially CPU-intensive, Head-Related Transfer Functions (HRTFs). To solve this, OpenSL ES allows an application to group two or more players that require exactly the same 3D properties using a 3D group. This is the second method for controlling a player's 3D properties.

It is not possible to use both of these methods at the same time: the 3D interfaces are exposed either on the player or on its 3D group.

4.4.1.2 When is a Player Rendered in 3D?

By default, players are not rendered in 3D; the application can still position the player on a one-dimensional line by setting its stereo position [see section 8.48], but no 3D effect should be applied to the player.

For a player to be rendered in 3D, it is necessary to expose a specific 3D interface during the player's creation. The specific interface that needs to be exposed is dependent on whether the player is to be in a 3D group.

In a standard case, when a player is not to be a member of a 3D group, the `SL3DLocationItf` interface [see section 8.5] is exposed on the player to signify that the player is to be rendered in 3D. As the presence of the `SL3DLocationItf` interface determines whether the player is 3D, this interface must be exposed on the player if any of the other 3D interfaces (`SL3DSourceItf` [see section 8.8], `SL3DDopplerItf` [see section 8.3], `SL3DMacroscopicItf` [see section 8.7]) are to be exposed on the player.

In the case when a player is to be a member of a 3D group, the `SL3DGroupingItf` interface should be exposed on the player. The player is treated as 3D and all subsequent 3D interfaces must be exposed on the 3D group, not on the player.

The `SL3DLocationItf` and `SL3DGroupingItf` interfaces are mutually exclusive. They cannot be exposed simultaneously by the same object. Any attempt to create an object that exposes both interfaces will fail.

The decision on whether to render the player in 3D is made at creation time, so it is not possible to add or remove either the `SL3DLocationItf` interface or `SL3DGrouping` interfaces dynamically using the `SLDynamicInterfaceManagementItf` interface [see section 8.17].

The term *3D source* is used in this specification to refer to both 3D positioned players and 3D groups.

4.4.2 3D Parameters

4.4.2.1 Coordinate System

All 3D vectors are expressed in right-handed Cartesian coordinates of the form (*x*, *y*, *z*), where *x* represents the x-axis position, *y* the y-axis position and *z* the z-axis position.

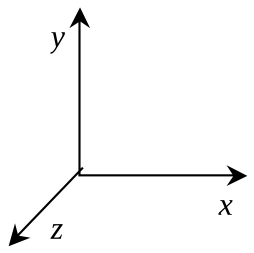


Figure 3: Coordinate system

4.4.2.2 Precision

Implementations of OpenSL ES are not required to support the full precision implied by the range of the parameter of the 3D functions. That is, a developer should not assume that minor changes in a parameter value will necessarily affect the output audio. By way of illustration: if the `SL3DLocationItf` interface method `SetLocation()` [see section 8.5] is used in order to change the 3D location of a 3D source by just 1 mm, it may actually generate bit-exact audio to that which would be generated if the 3D source had not been moved, as internally the implementation might support, say, 28 bits of precision rather than the full 32 bits implied by the type of the vector.

Even when an implementation does not internally support full precision, the implementation shall still return the last *set* value in a *get* query for the same parameter, not a rounded value.

4.4.2.3 Interactivity

OpenSL ES is designed for interactive applications. For this reason, there should be no discernable audio glitches when changing any of the 3D parameters.

4.4.3 Levels of 3D Sources

The perceived loudness of a 3D source within the minimum distance (specified using `SL3DSourceItf::SetRolloffDistances()` [see section 8.8]) is equal to the perceived loudness of a non-3D stereo source with gain of -3 dB. This applies when the same typical program material is used as input for both sources such that the 3D source has mono material as input and the stereo source has the same material duplicated in both its channels, and no additional effects (such as orientation or reverb) are used for either source.

4.4.4 3D Positioning Multi-Channel Players

In most cases, there is no benefit gained in 3D positioning multi-channel data (such as stereo) over mono data. However, some implementations may take advantage of the multi-channel data when rendering a player's macroscopic behavior [see section 8.7]. Implementations shall support exposing the 3D interfaces on a player whose data source is multiple-channel. The exact behavior is undefined, but all channels in the input data should be represented in the 3D positioned sound source.

4.4.4.1 3D and MIDI Players

A MIDI player can be 3D positioned in the same way as a sampled audio player. In such cases it is the output from the synthesizer that is 3D positioned.

4.4.5 Object Relationships

The relationship between the two 3D audio objects (listener [see section 7.6] and 3D groups [see section 7.1]) and the engine and players is summarized in the UML diagram in Figure 4.

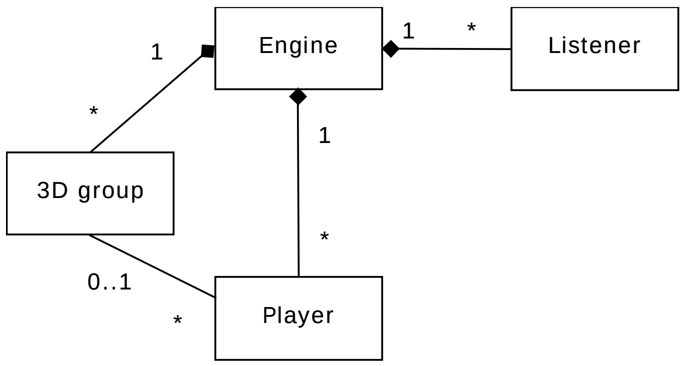


Figure 4: Object relationships

An engine creates multiple players, 3D groups and listeners. A player may be a member of one 3D group at a time and many players may be members of the same 3D group.

4.5 Effects

OpenSL ES supports various audio effects, from pan control to advanced 3D audio and environmental reverberation. Rudimentary effects such as panning are handled using the `SLVolumeItf` interface [see section 8.48] and 3D audio is controlled using its own set of interfaces [see section 4.4]. This section documents the remaining effects: bass boost, equalization, reverberation and virtualization.

4.5.1 Effects Architecture

Effects are handled in the same way as the other controls in the API and as such an effect is controlled using an interface exposed on an object. Exposing an effect interface on different object types will result in the effect being applied to different points in the audio processing chain.

Global effects are exposed via interfaces on an Output Mix object and behave as global insert effects on the output mix. It is not necessary to use the effect send interface for the effect to be audible.

Auxiliary effects, of which the only standardized auxiliary effect is reverb, are also exposed via interfaces on an Output Mix object. For a player to contribute to the auxiliary effect, it is necessary to expose the effect send interface [see section 8.17] on the players for the effect to be applied.

OpenSL ES does not mandate any per-player effects. However, where supported they should be exposed as interfaces directly on the player.

4.5.2 Bass Boost

Bass boost is an audio effect to boost or amplify low frequencies of the sound. OpenSL ES supports bass boost via `SLBassBoostItf` [see section 8.13].

4.5.3 Equalization

Equalizer is an audio filter to change the frequency envelope of the sound. OpenSL ES supports equalizer via `SLEqualizerItf` [see section 8.24].

4.5.4 Virtualization

OpenSL ES supports audio virtualizer functionality. Audio virtualization is a general name for an effect to virtualize audio channels. Typically this effect is used in order to compensate the mismatch between the loudspeaker setup used when producing the audio and the audio output device used when the audio is played back by the end-user. The exact behavior of this effect is dependent on the number of audio input channels and the type and number of audio output channels of the device. For example, in the case of a stereo input and stereo headphone output, a stereo widening is used when this effect is turned on. An input with 5.1-channels similarly uses some virtual surround algorithm.

OpenSL ES supports virtualization via `SLVirtualizerItf` [see section 8.46].

4.5.5 Reverberation

A sound generated within a room travels in many directions. The listener first hears the direct sound from the source itself. Later, he or she hears discrete echoes caused by sound bouncing off nearby walls, the ceiling and the floor. As sound waves arrive after undergoing more and more reflections, individual reflections become indistinguishable and the listener hears continuous reverberation that decays over time.

Reverb is vital for modeling a listener’s environment. It can be used in music applications to simulate music being played back in various environments, or in games to immerse the listener within the game’s environment.

OpenSL ES supports one global reverb environment that can be controlled by two different interfaces, one supporting a fixed set of simple presets and another supporting a more advanced interactive interface.

The preset-based interface is most suited to music applications where the environment may be selected within the user interface. OpenSL ES supports a preset reverb via the `SLPresetReverbItf` interface [see section 8.40].

The more advanced environmental interface is suited to game applications, where the environment may change as a listener moves around. OpenSL ES supports an environmental reverb via the `SLEnvironmentalReverbItf` interface [see section 8.23].

Reverb is an auxiliary effect, so only selected players have reverb applied. For a player to include reverb, it is necessary to expose the `SLEffectSendItf` interface [see section 8.17] and expose one of the reverb interfaces on the Output Mix.

4.6 Example Use Cases

This section illustrates the use of OpenSL ES objects and interfaces in some typical audio usage scenarios.

4.6.1 Sampled Audio Playback

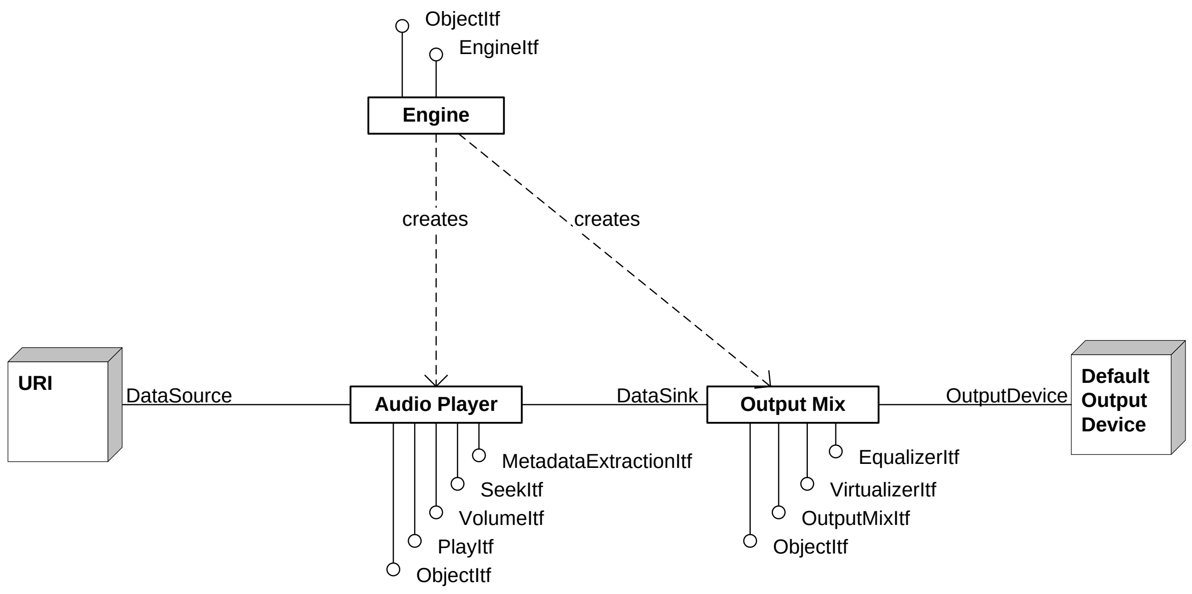


Figure 5: Sampled audio playback use case

An Audio Player object [see section 7.2] is used for sampled audio playback. An Audio Player is created using the `SLEngineItf` interface of the engine object. Upon creation, we associate the Audio Player with an Output Mix [see section 7.9] (which we create using the `SLEngineItf` interface [see section 8.21]) for audio output. We also set the data source of the Audio Player during creation. The data source could be, for example, a URI pointing to an audio file in the local file system. The Output Mix is by default associated with the system-dependent default output device.

4.6.2 MIDI Playback

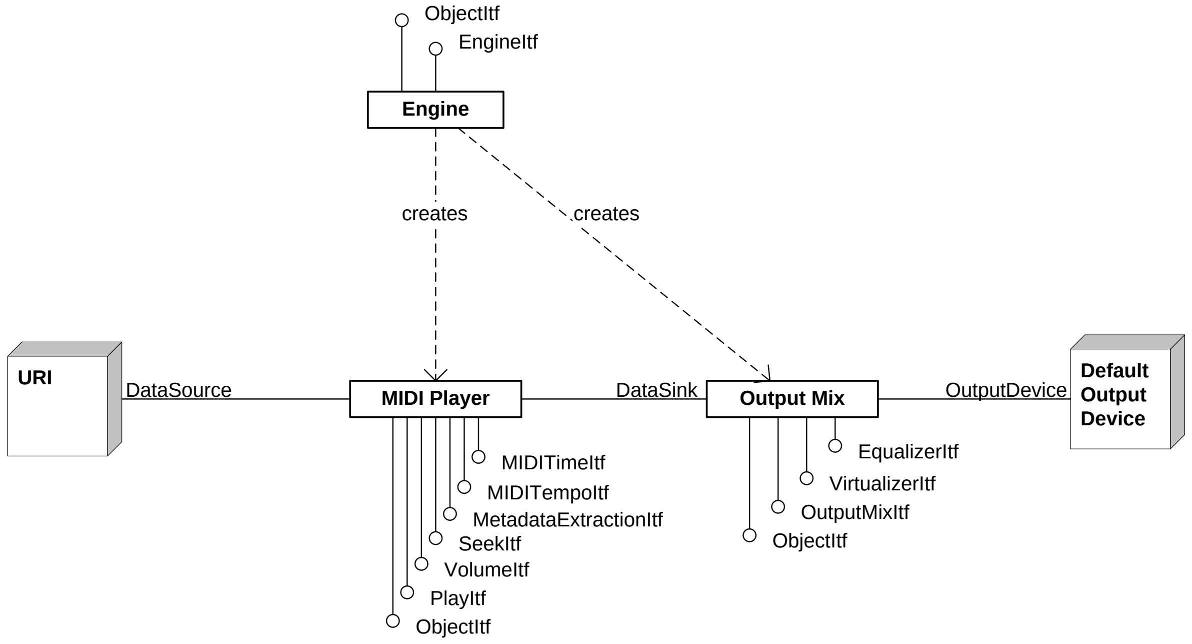


Figure 6: MIDI playback use case

MIDI playback is accomplished in a similar mannar to sampled audio playback, as described in the previous section. The only difference is that an MIDI Player object [see section 7.8] is used for playback instead of an Audio Player object.

4.6.3 3D Audio

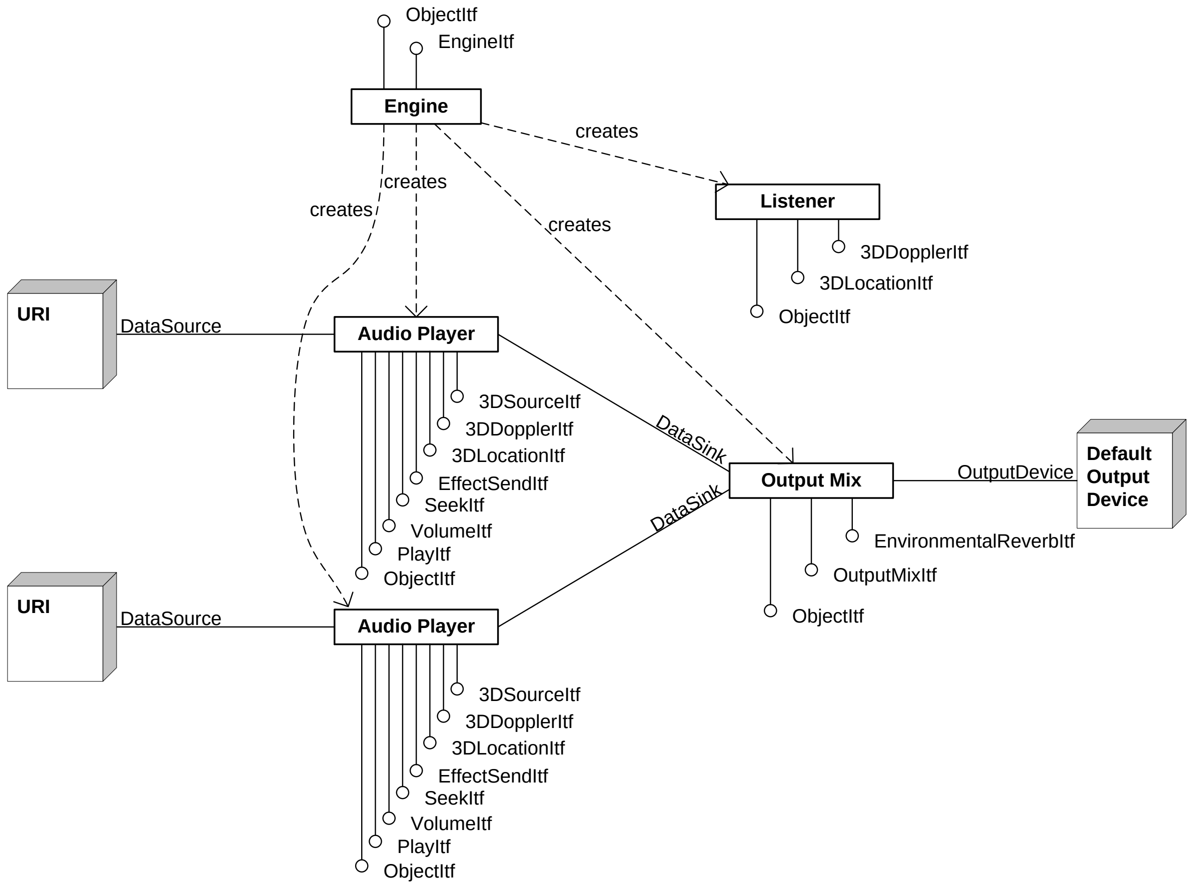


Figure 7: 3D audio use case

This use case illustrates positional 3D audio rendering and use of two sampled audio players simultaneously. Both Audio Player objects [see section 7.2] are created using the `SLEngineItf` interface [see section 8.21] of the engine object. Upon creation, we associate both Audio Players with the same Output Mix [see section 7.9] (which we also create with the `SLEngineItf` interface) for audio output.

Requesting `SL3DLocationItf` interfaces [see section 8.5] from the Audio Players upon their creation causes them to be rendered as 3D sources. The virtual listener is controlled with a Listener object [see section 7.6] which we also create using the `SLEngineItf` interface of the engine object.

The reverberation of the virtual acoustical space is controlled by `SLEnvironmentalReverbItf` interface [see section 8.23] of the Output Mix. `SLEffectSendItf` interfaces [see section 8.19] are exposed on the Audio Players to feed their audio signal to the reverberator of the Output Mix.

4.6.4 Recording Audio

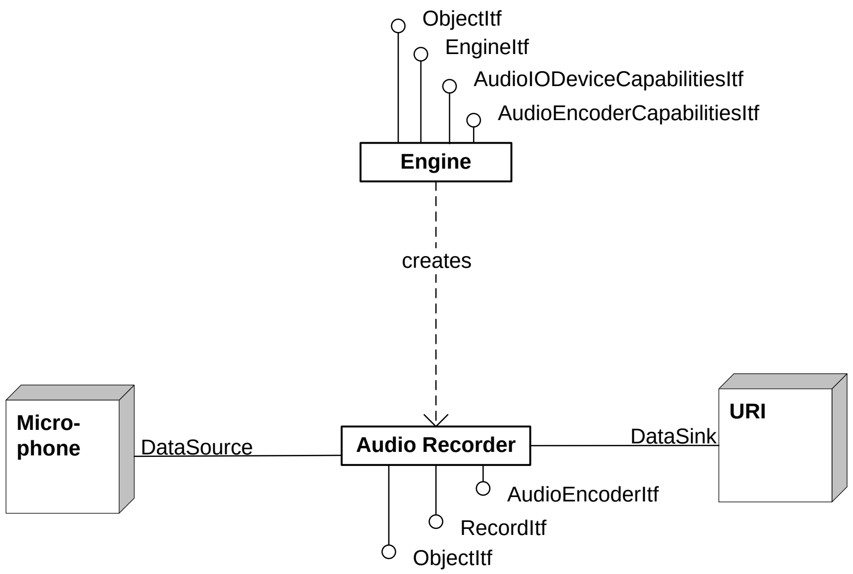


Figure 8: Recording audio use case

An Audio Recording use case is handled by an Audio Recorder object [see section 7.3]. We create the Audio Recorder object using `SLEngineItf` interface of the engine object. Upon creation, we associate it with an audio data source, which can be, for example, a microphone (an audio input device). The data sink of the Audio Recorder can be a URI pointing to an audio file in the local file system to which the audio will be recorded.

4.6.5 Reading Metadata

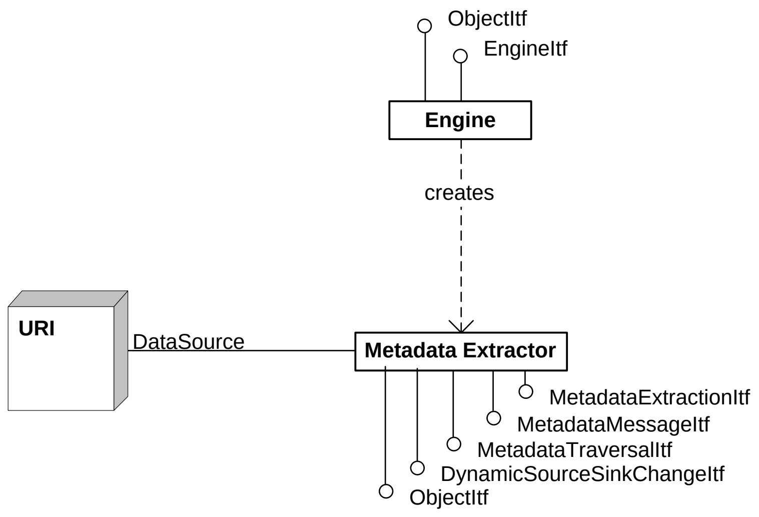


Figure 9: Reading metadata use case

A Metadata Extractor object [see section 7.7] will read the metadata of an audio file without allocating resources for audio playback. As in other use-cases, we create the object using the `SLEngineItf` interface of the engine object and upon creation, we set the data source of the Metadata Extractor. The data source is typically a URI pointing to an audio file in the local file system. However, the Metadata Extractor supports the `SLDynamicSourceSinkChangeItf` interface [see section 8.19] which we can use to change the data source. Therefore we may extract metadata from multiple files (in series) without creating a new Metadata Extractor object for every single file. The `SLMetadataExtractionItf` and `SLMetadataTraversalItf` interfaces [see sections 8.26 and 0] are used for actually reading and traversing the metadata from a file. The SLMetadataMessageItf is used to set callbacks that execute whenever a metadata item is encountered.

4.7 Minimum Requirements

Minimum requirements determine what functionality can be relied on between implementations, which helps write more portable applications. Section 7 introduces the concept of a *mandated interface* and for each profile defines the interfaces that shall be supported for each object type. This gives the application developer a guarantee of what interfaces are supported on each object but in practice resource limitations will limit the number of concurrent objects that can be supported at a given time. In order to strengthen the guarantee of what is supported in all implementations for each profile we define a set of use-cases that must be supported.

The use-cases are defined using diagrams each of which specifies many possible use-cases. Implementations must support all the different choices that are presented in the diagram. These are highlighted using “OR” and “/”. For example, the use-case below shows that a conformant implementation must support a use case with an audio player with a “Buffer Source (Type 1)” data source and a separate use-case with an audio player with a “Buffer Queue Source (Type 1)” data source. The diagram also shows that the implementation must support use-cases where `SL3DDopplerItf` is exposed and use-cases where `SLRatePitchItf` is exposed.

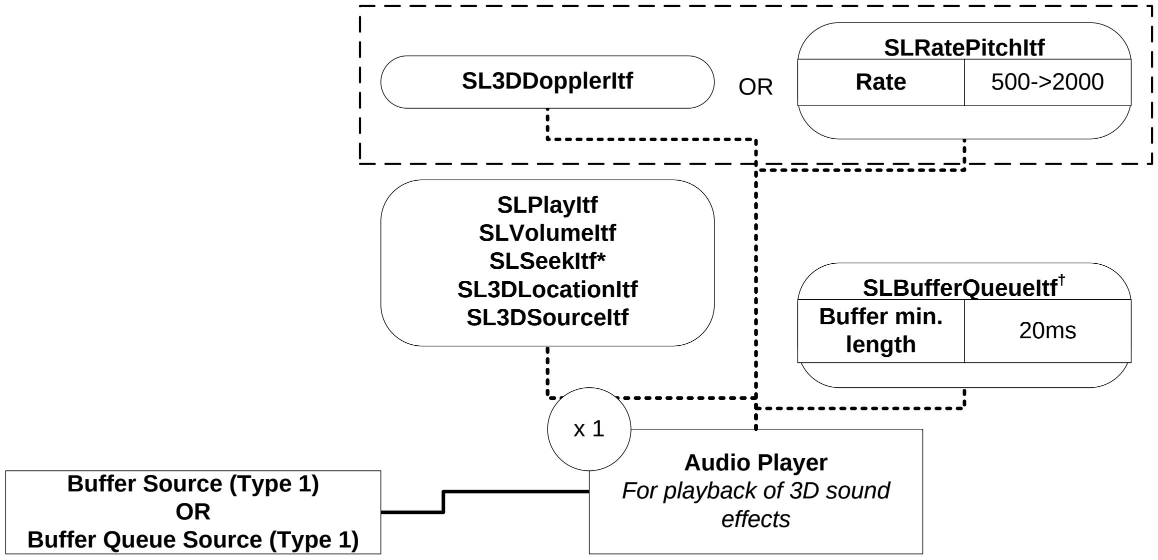


Figure 10: Example use case diagram

In addition to supporting the exact use-cases shown in each diagram, an implementation must support all the use-cases that can be formed by removing any non-implicit interfaces and objects, as long as no other changes are made. For example, from the use-case diagram in Figure 10 it can be determined that the implementation must support a use-case with an audio player with only the `SLPlayItf` interface exposed.

Unless specified otherwise, an implementation must support the whole range of parameters for an interface exposed on an object. Where only a subset of parameters needs to be supported, this is documented in the use-case or explicitly in the specification. For example, the implementation need only support rates from 500 to 2000 ‰ in the use-case specified in Figure 10.

4.7.1 Phone Profile

4.7.1.1 Use Case 1

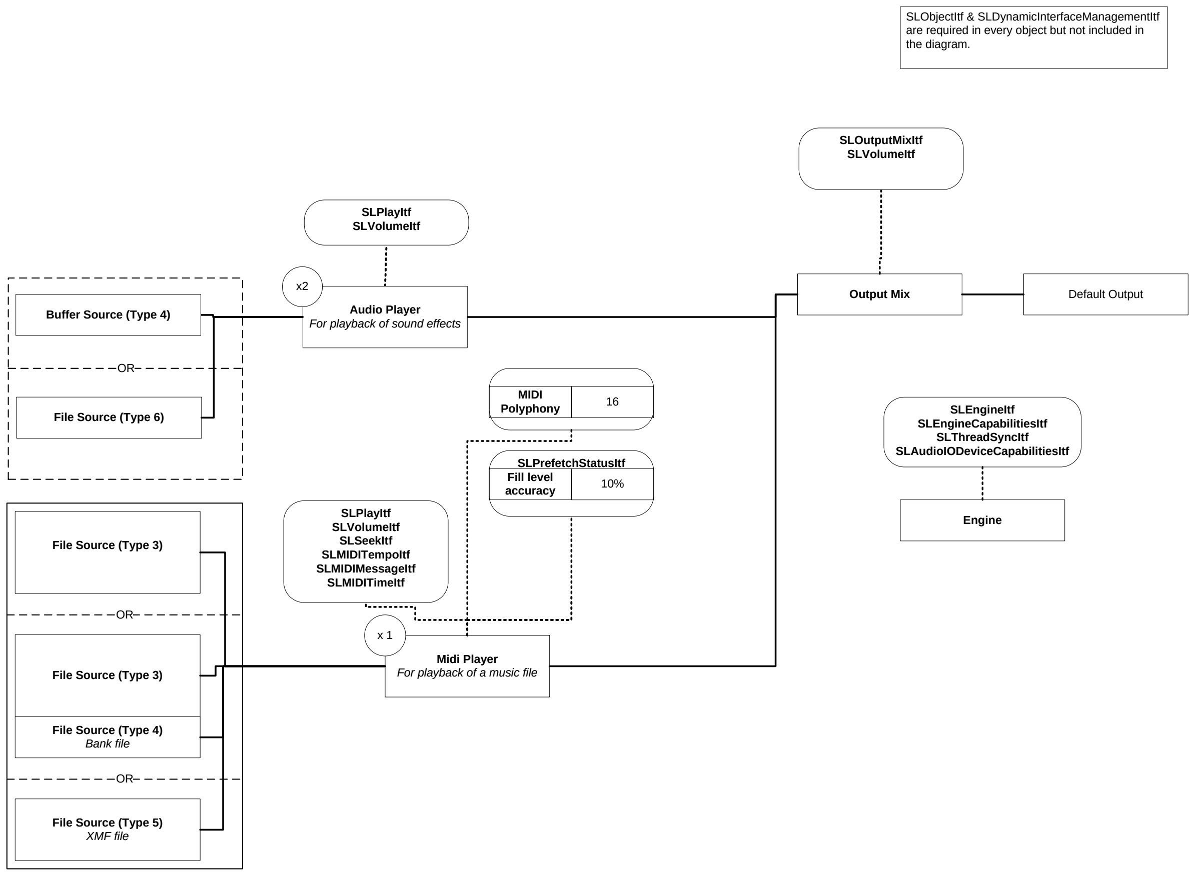


Figure 11: Phone profile – use case 1

4.7.1.2 Use Case 2

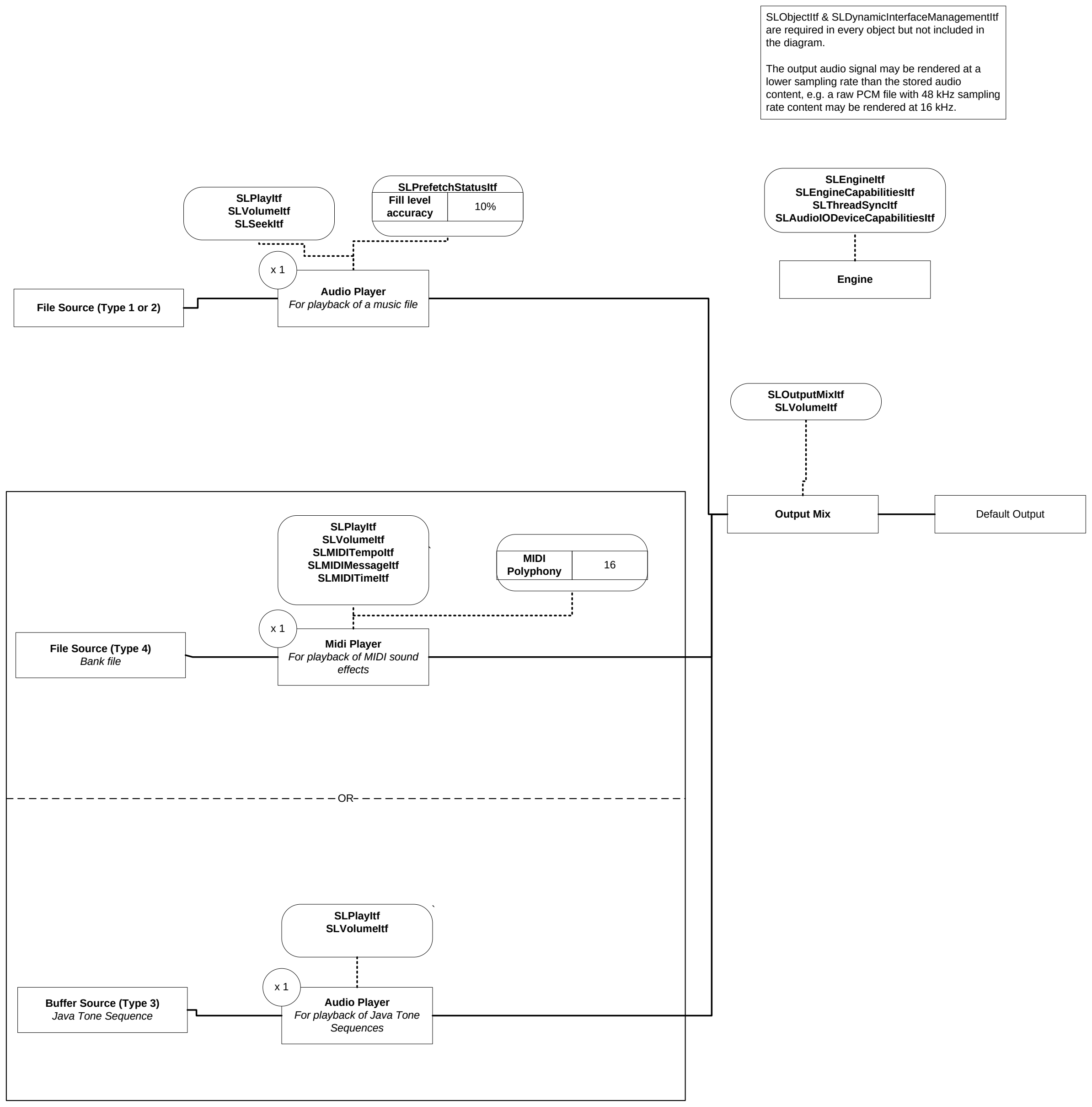

SLObjectItf & SLDynamicInterfaceManagementItf are required in every object but not included in the diagram.
The output audio signal may be rendered at a lower sampling rate than the stored audio content, e.g. a raw PCM file with 48 kHz sampling rate content may be rendered at 16 kHz.
SLPlayItf
SLVolumeItf
SLSeekItf
SLPrefetchStatusItf
Fill level accuracy
10%
SLEngineItf
SLEngineCapabilitiesItf
SLThreadSyncItf
SLAudioIODeviceCapabilitiesItf
Engine
x 1
Audio Player
For playback of a music file
File Source (Type 1 or 2)
SLOutputMixItf
SLVolumeItf
Output Mix
Default Output
SLPlayItf
SLVolumeItf
SLMIDITempoItf
SLMIDIMessageItf
SLMIDITimeItf
MIDI Polyphony
16
x 1
Midi Player
For playback of MIDI sound effects
File Source (Type 4)
Bank file
OR
SLPlayItf
SLVolumeItf
x 1
Audio Player
For playback of Java Tone Sequences
Buffer Source (Type 3)
Java Tone Sequence

Figure 12: Phone profile – use case 2

4.7.1.3 Use Case 3

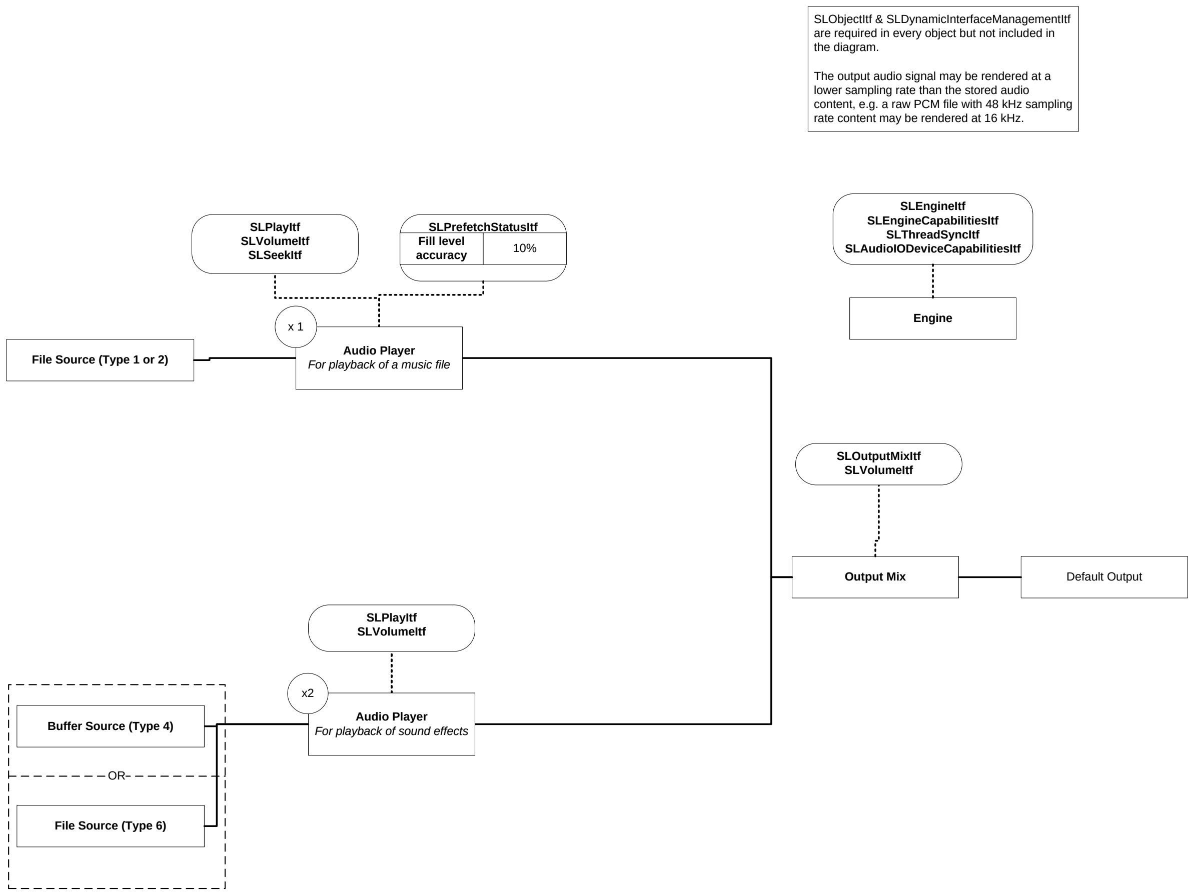
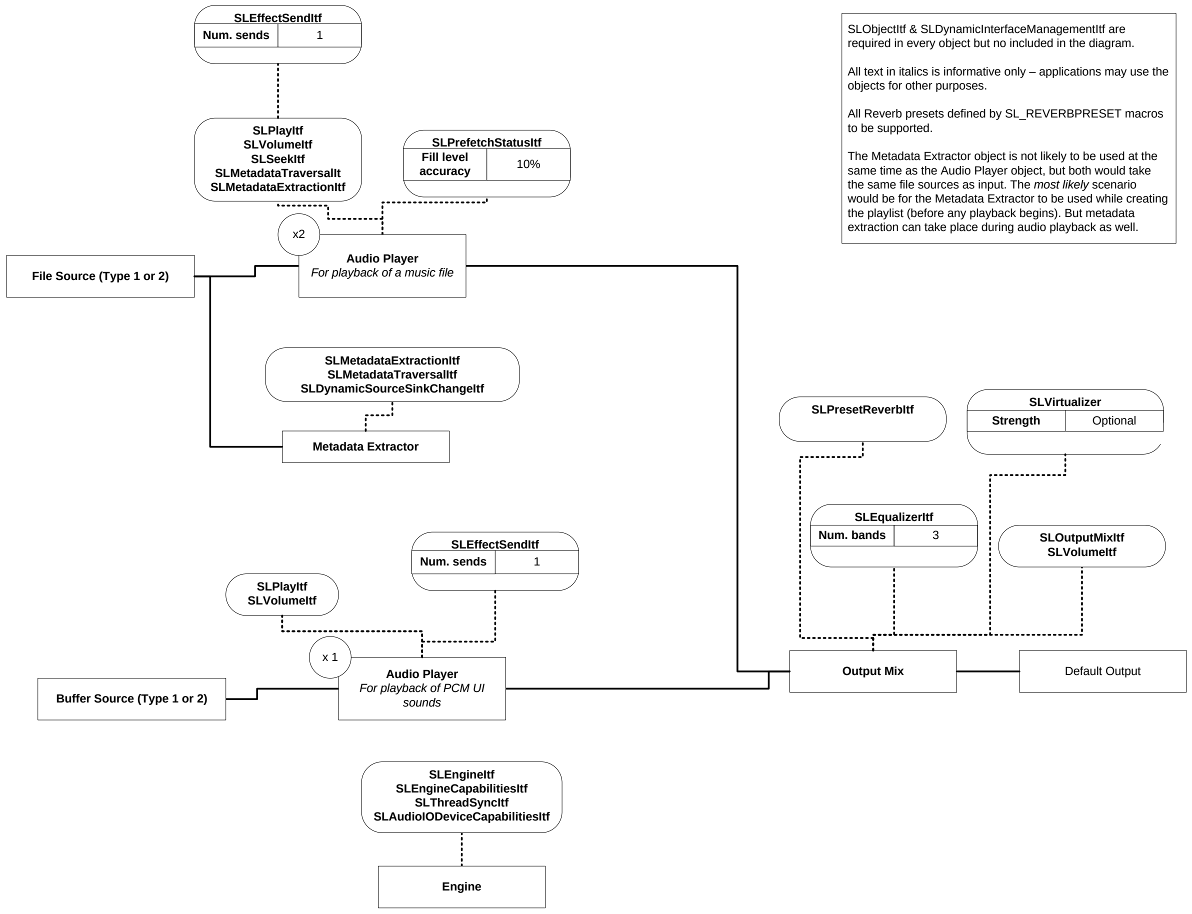

SLObjectItf & SLDynamicInterfaceManagementItf are required in every object but not included in the diagram.
The output audio signal may be rendered at a lower sampling rate than the stored audio content, e.g. a raw PCM file with 48 kHz sampling rate content may be rendered at 16 kHz.
SLPlayItf
SLVolumeItf
SLSeekItf
SLPrefetchStatusItf
Fill level accuracy
10%
SLEngineItf
SLEngineCapabilitiesItf
SLThreadSyncItf
SLAudioIODeviceCapabilitiesItf
Engine
x 1
File Source (Type 1 or 2)
Audio Player
For playback of a music file
SLOutputMixItf
SLVolumeItf
Output Mix
Default Output
SLPlayItf
SLVolumeItf
x2
Buffer Source (Type 4)
OR
File Source (Type 6)
Audio Player
For playback of sound effects

Figure 13: Phone profile – use case 3

4.7.2 Music Profile

4.7.2.1 Use Case 1



SLObjectItf & SLDynamicInterfaceManagementItf are required in every object but no included in the diagram.

All text in italics is informative only – applications may use the objects for other purposes.

All Reverb presets defined by SL_REVERBPRESET macros to be supported.

The Metadata Extractor object is not likely to be used at the same time as the Audio Player object, but both would take the same file sources as input. The *most likely* scenario would be for the Metadata Extractor to be used while creating the playlist (before any playback begins). But metadata extraction can take place during audio playback as well.

Figure 14: Music profile – use case 1

4.7.3 Game Profile

4.7.3.1 Use Case 1

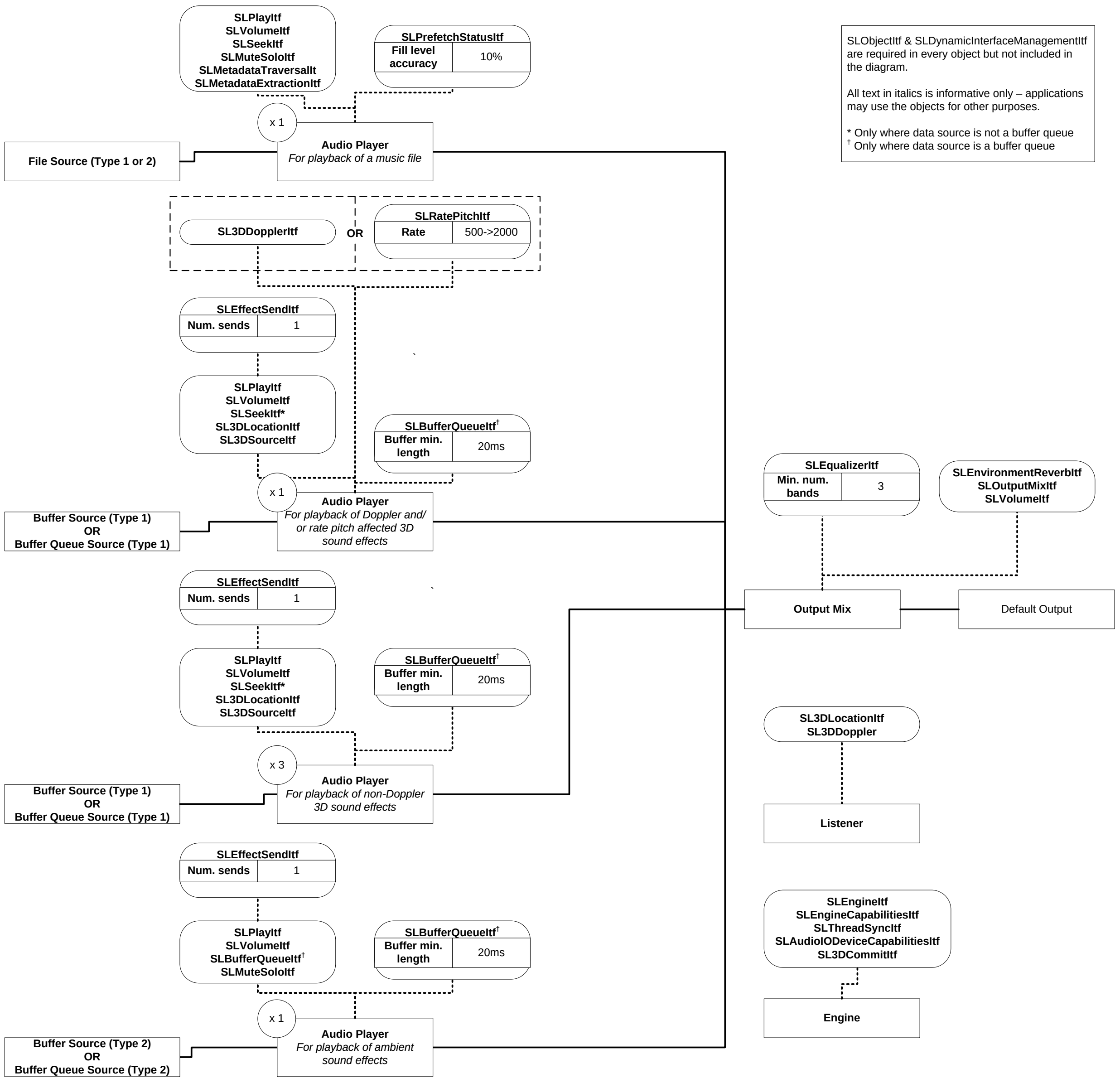


Figure 15: Game profile – use case 1

4.7.3.2 Use Case 2

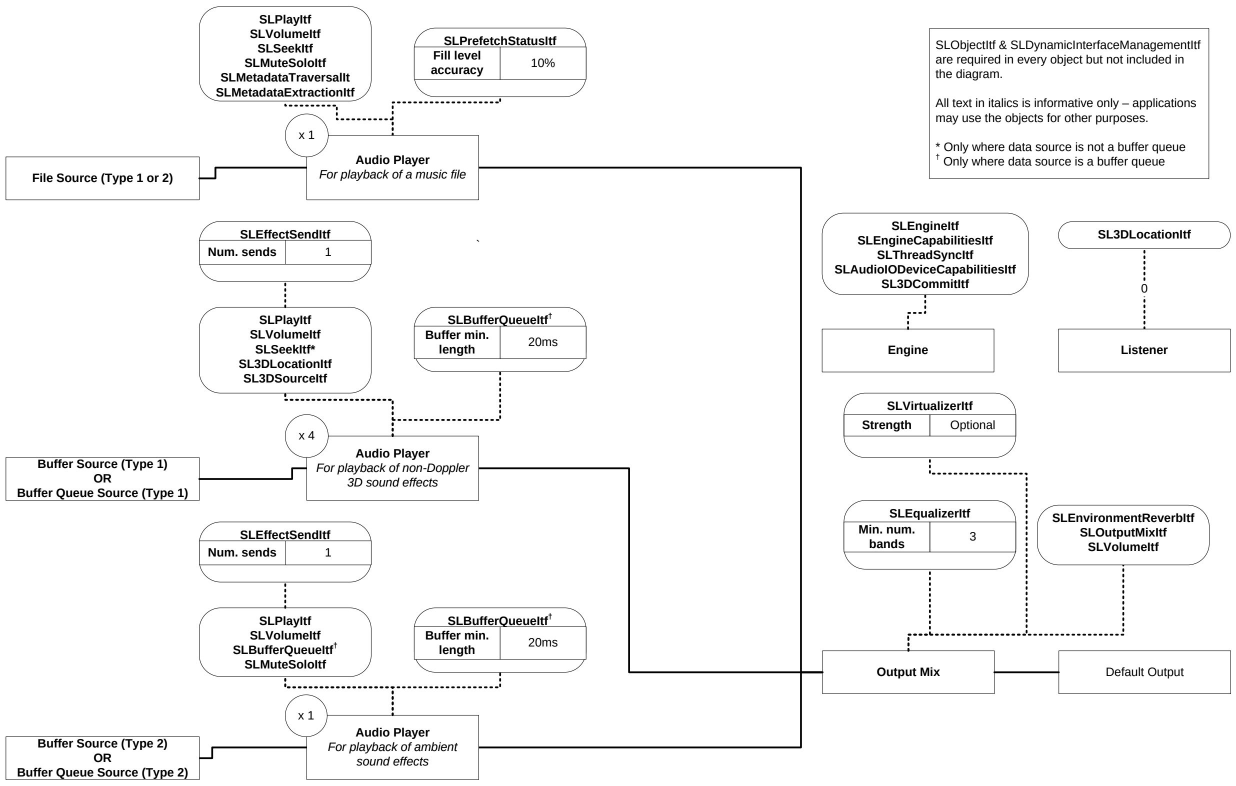


Figure 16: Game profile – use case 2

4.7.3.3 Use Case 3

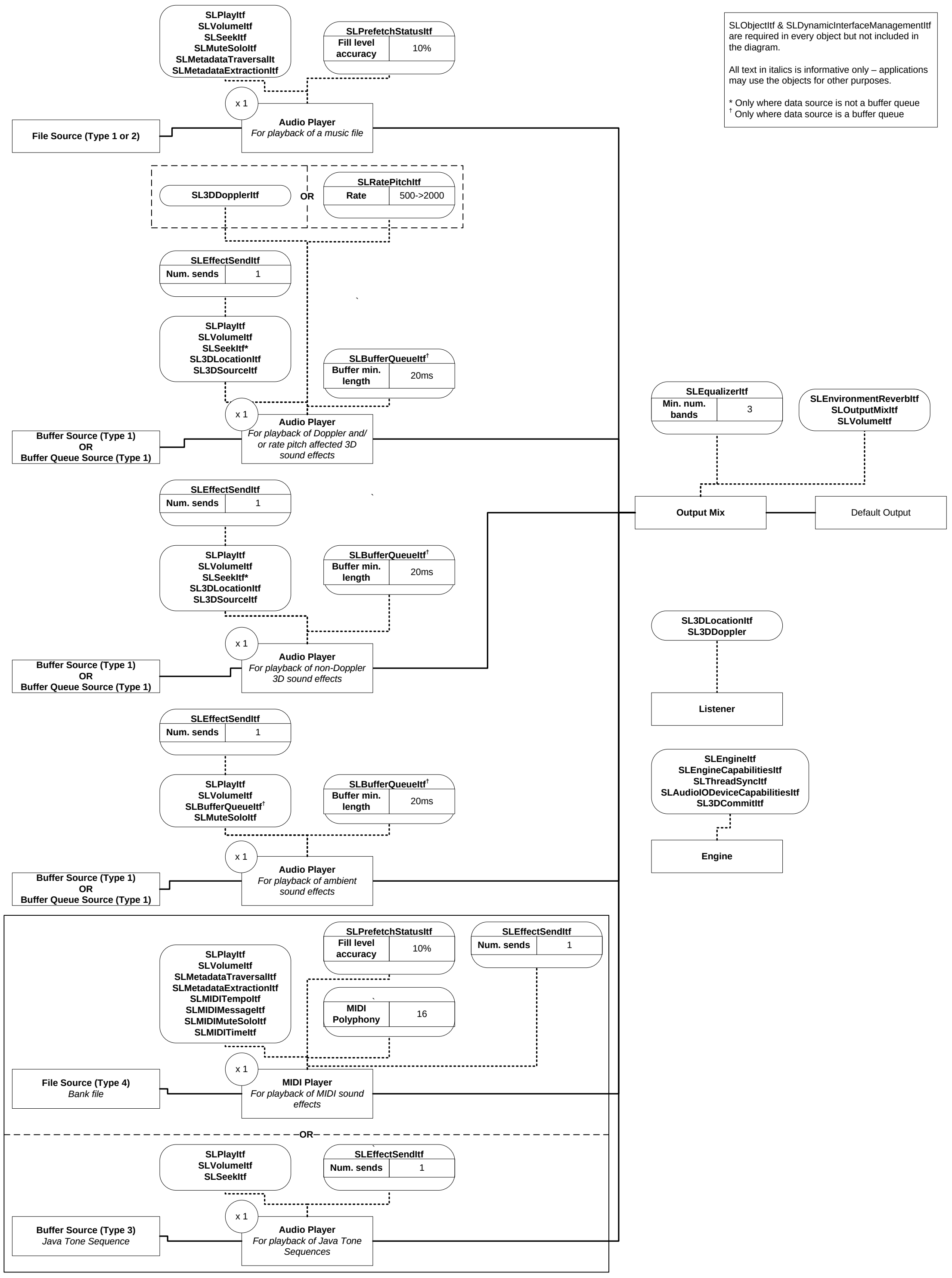


Figure 17: Game profile – use case 3

4.7.3.4 Use Case 4

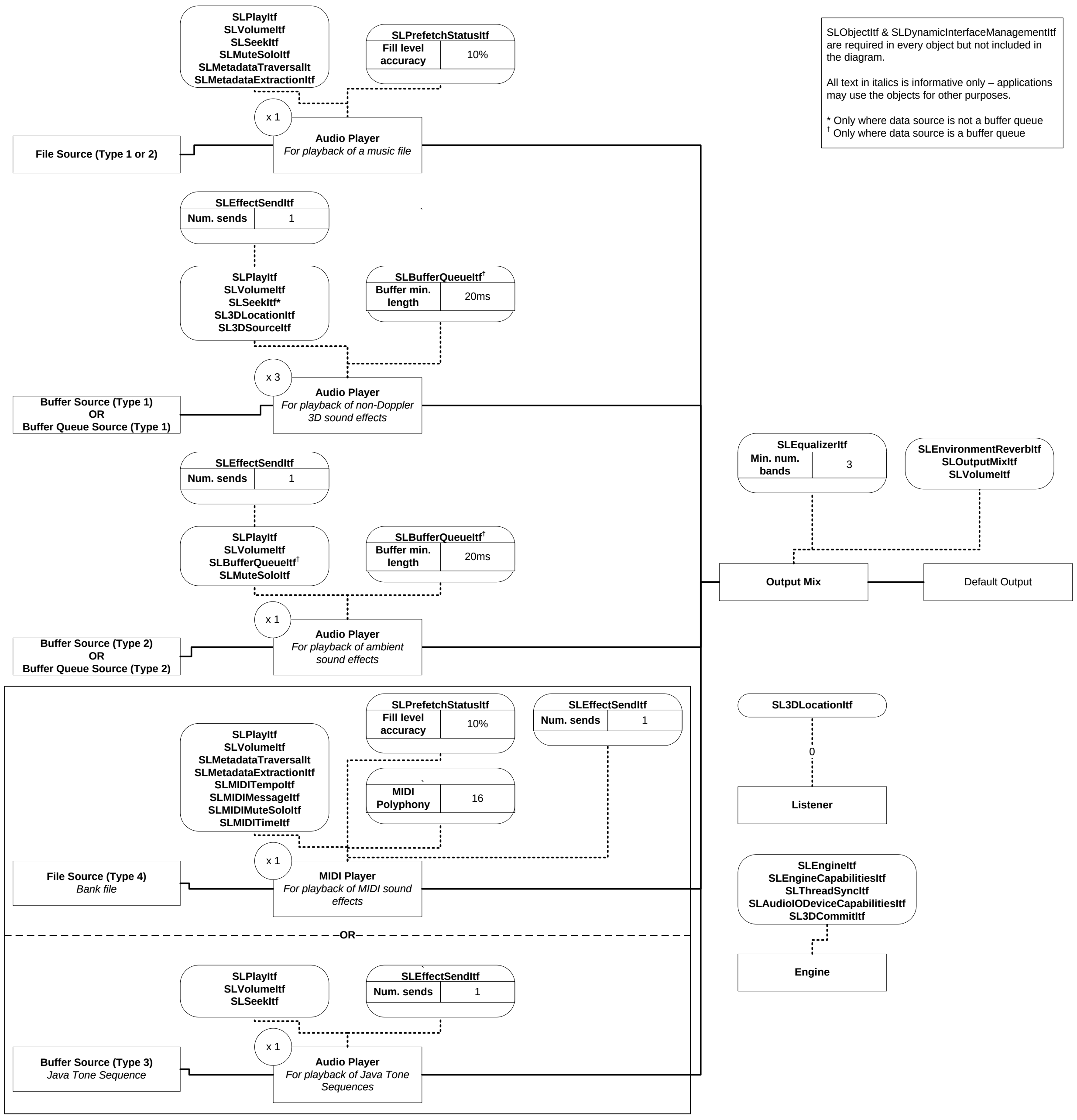


Figure 18: Game profile – use case 4

4.7.3.5 Use Case 5

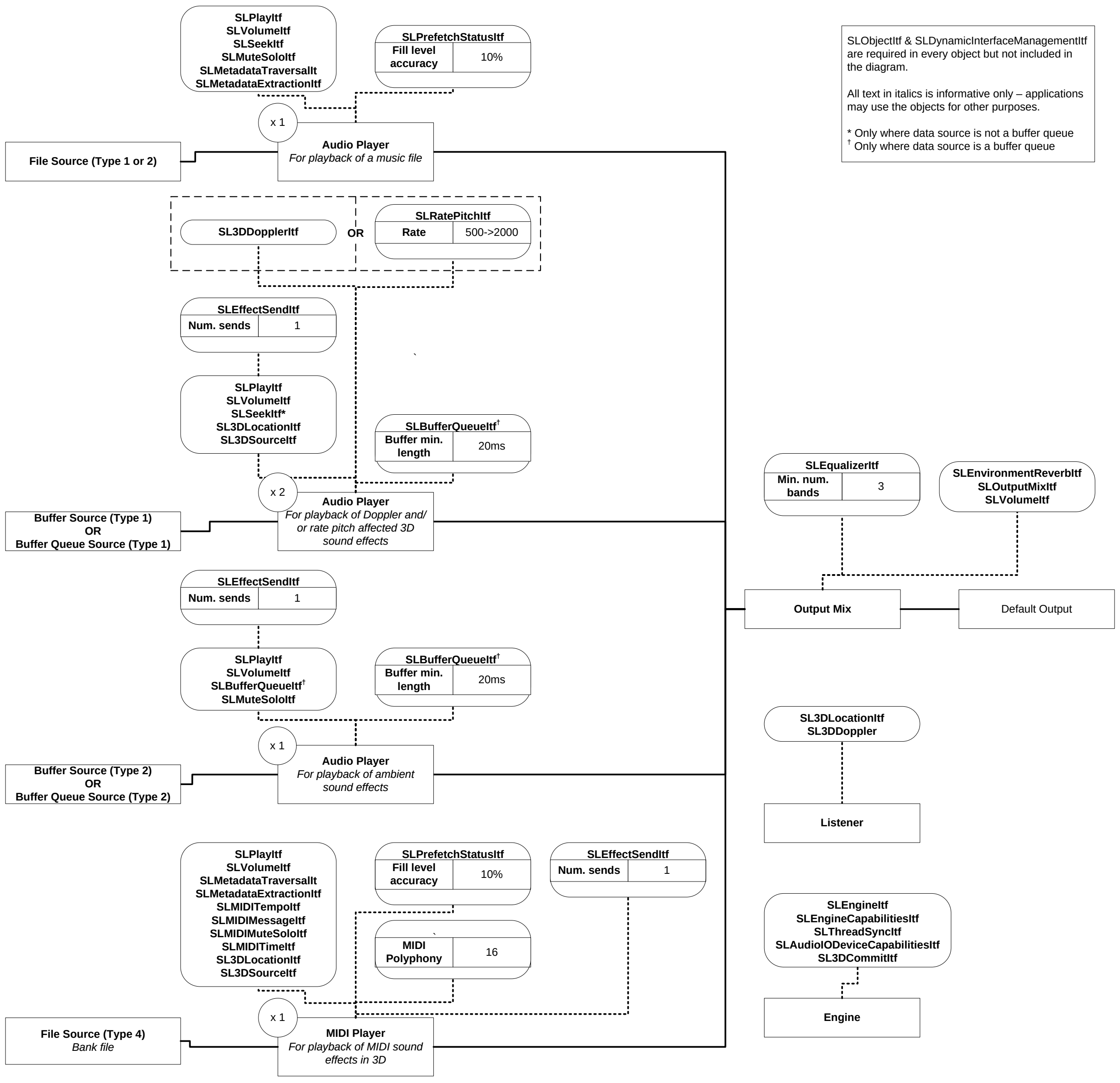
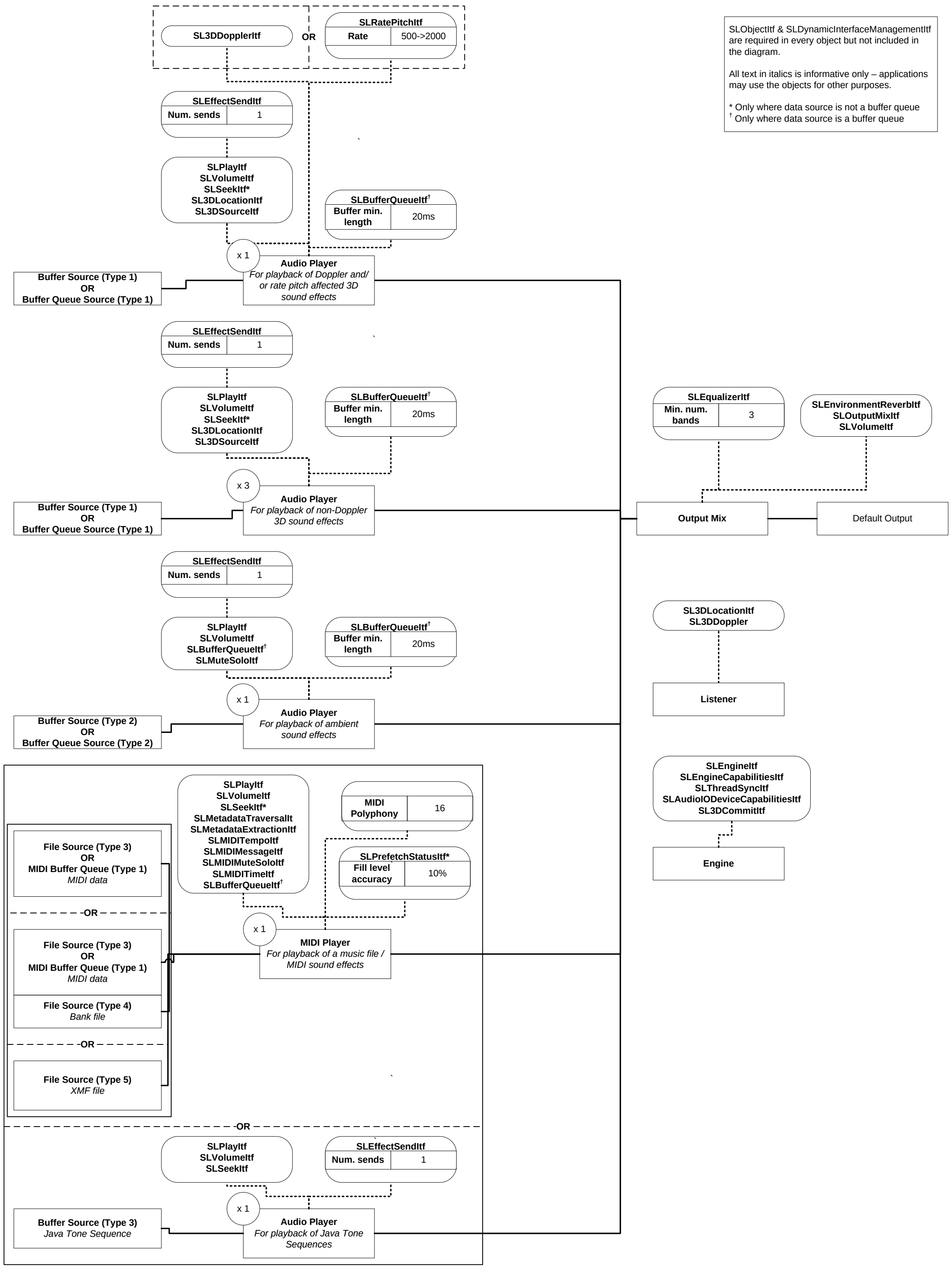


Figure 19: Game profile – use case 5

4.7.3.6 Use Case 6


SLObjectItf & SLDynamicInterfaceManagementItf are required in every object but not included in the diagram.

All text in italics is informative only – applications may use the objects for other purposes.

* Only where data source is not a buffer queue
[†] Only where data source is a buffer queue

Figure 20: Game profile – use case 6

4.7.3.7 Use Case 7

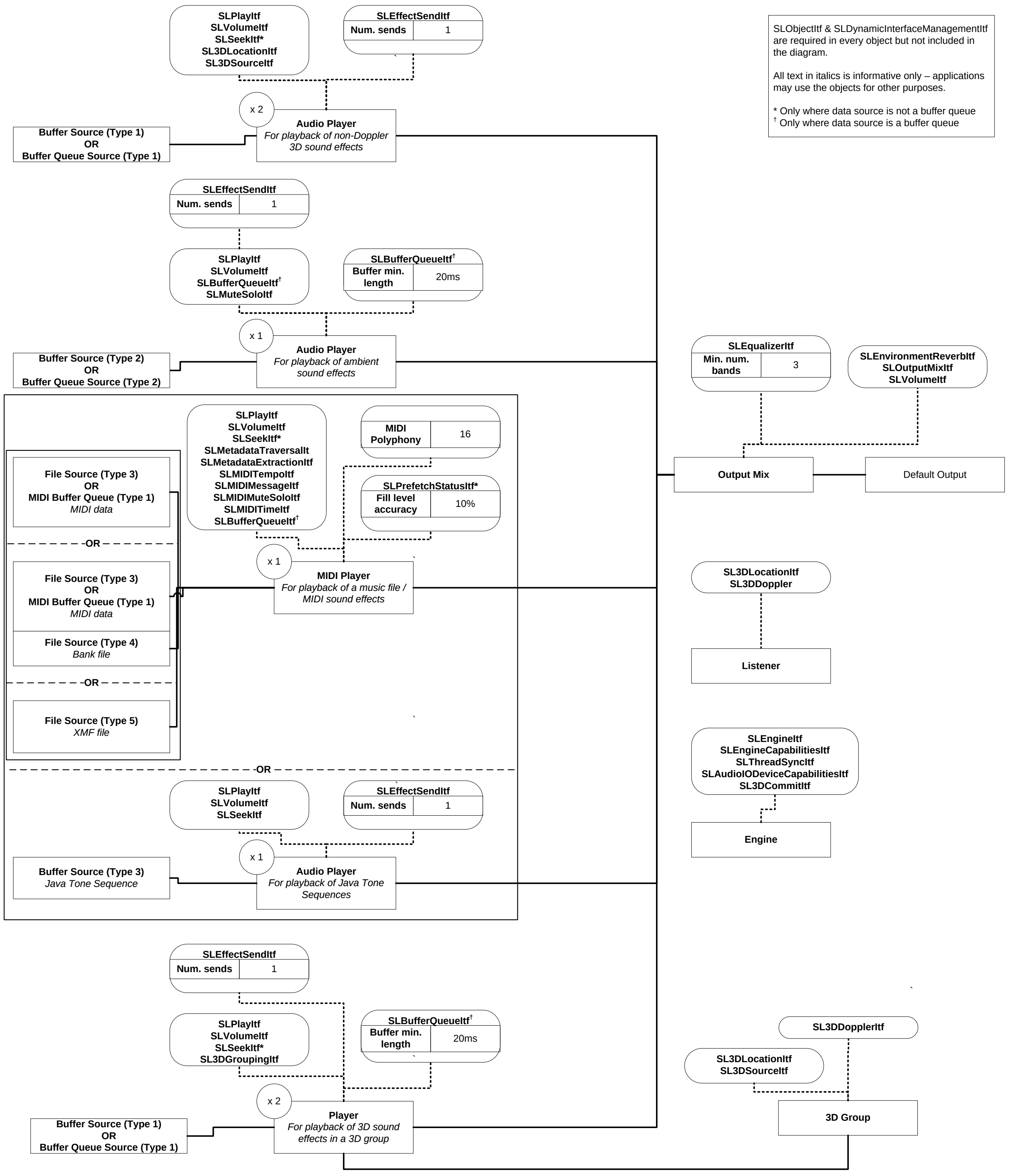


Figure 21: Game profile – use case 7

4.7.3.8 Use Case 8

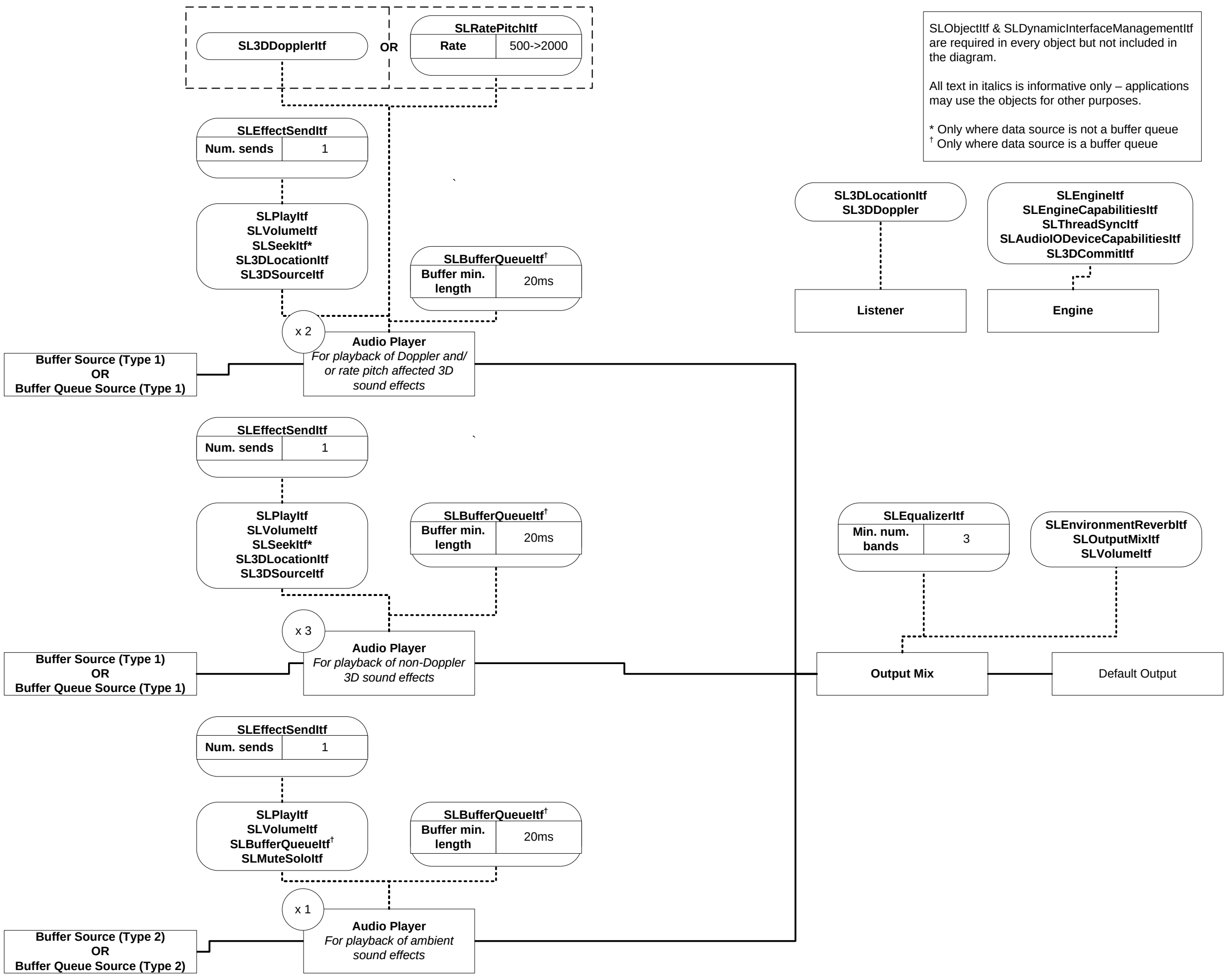


Figure 22: Game profile – use case 8

4.7.4 Data Sources

File sources

File Source (Type 1)	
Location	File System
Format Type	MIME
MIME Type	WAV(PCM)
Channels	1/2
Samples/Sec	8/16/22.05/24/ 32/44.1/48 K
Bits/Sample (Container Size)	8(8)/16(16)
Channel Mask	None

File Source (Type 2)	
Location	File System
Format Type	PCM
Channels	1/2
Samples/Sec	8/16/22.05/24/ 32/44.1/48 K
Bits/Sample (Container Size)	8(8)/16(16)
Channel Mask	None
Byte Order	Native

File Source (Type 3)	
Location	File System
Format Type	MIME
MIME Type	MIDI (SP-MIDI)
Min SMF tracks (MTRk chunks)	16

File Source (Type 4)	
Location	File System
Format Type	MIME
MIME Type	DLS (Mobile DLS)

File Source (Type 5)	
Location	File System
Format Type	MIME
MIME Type	Mobile XMF
Min SMF tracks (MTRk chunks)	16

File Source (Type 6)	
Location	File System
Format Type	PCM
Channels	1/2
Samples/Sec	8/16 K
Bits/Sample (Container Size)	8(8)/16(16)
Channel Mask	None
Byte Order	Native

Buffer sources

Buffer Source (Type 1)	
Location	Memory
Format Type	PCM
Channels	1
Samples/Sec	8/16/22.05/24 K
Bits/Sample (Container Size)	8(8)/16(16)
Channel Mask	None
Byte Order	Native

Buffer Source (Type 2)	
Location	Memory
Format Type	PCM
Channels	1/2
Samples/Sec	8/16/22.05/24 K
Bits/Sample (Container Size)	8(8)/16(16)
Channel Mask	None
Byte Order	Native

Buffer Source (Type 3)	
Location	Memory
Format Type	MIME
MIME Type	audio/x-tone-seq

Buffer Source (Type 4)	
Location	Memory
Format Type	PCM
Channels	1/2
Samples/Sec	8/16 K
Bits/Sample (Container Size)	8(8)/16(16)
Channel Mask	None
Byte Order	Native

Buffer queue sources

Buffer Queue Source (Type 1)	
Location	Buffer Queue
Num Buffers	8
Format Type	PCM
Channels	1
Samples/Sec	8/16/22.05/24 K
Bits/Sample (Container Size)	8(8)/16(16)
Channel Mask	None
Byte Order	Native

Buffer Queue Source (Type 2)	
Location	Buffer Queue
Num Buffers	8
Format Type	PCM
Channels	1/2
Samples/Sec	8/16/22.05/24 K
Bits/Sample (Container Size)	8(8)/16(16)
Channel Mask	None
Byte Order	Native

MIDI Buffer queue sources

Buffer Queue Source (Type 1)	
Location	Buffer Queue
Num Buffers	8
Format Type	MIME
MIME Type	MIDI

Figure 23: Data sources

PART 2: API REFERENCE

5 Base Types and Units

OpenSL ES defines a set of cross-platform fixed width types that are used within the API. The definition of these are system-dependent and the platform provider must specify these types. OpenSL ES also defines a set of types for different units required by the API, such as distance and volume. To aide programmability, most of these units are based on the thousandth unit of a SI unit [ISO1000].

5.1 Standard Units

The table below shows the standard types for units used in OpenSL ES.

Table 8: OpenSL ES Units Types

Unit	Measurement	C type
Volume level	millibel (mB)	`SLmillibel`
Time	millisecond (ms)	`SLmillisecond`
Frequency	milliHertz (mHz)	`SLmilliHertz`
Distance	millimeter (mm)	`SLmillimeter`
Angle	millidegree (mdeg)	`SLmillidegree`
Scale/Factor	permille (‰)	`SLpermille`

5.2 Base Types

```
typedef <system dependent> SLint8;
typedef <system dependent> SLuint8;
typedef <system dependent> SLint16;
typedef <system dependent> SLuint16;
typedef <system dependent> SLint32;
typedef <system dependent> SLuint32;
typedef <system dependent> SLfloat32;
typedef <system dependent> Slfloat64;
typedef <system dependent> SLchar;
typedef SLuint32           SLboolean;
typedef SLint16            SLmillibel;
typedef SLuint32           SLmillisecond;
typedef SLuint32           SLmilliHertz;
typedef SLint32            SLmillimeter;
typedef SLint32            SLmillidegree;
typedef SLint16            SLpermille;
typedef SLuint32           SLmicrosecond;
typedef SLuint32           SLresult;
```

Table 9: Base Types

Type	Description
SLint8	An 8-bit signed type. The definition of this type is system-dependent.
SLuint8	An 8-bit unsigned type. The definition of this type is system-dependent.
SLint16	A 16-bit signed type. The definition of this type is system-dependent.
SLuint16	A 16-bit unsigned type. The definition of this type is system-dependent.
SLint32	A 32-bit signed type. The definition of this type is system-dependent.
SLuint32	A 32-bit unsigned type. The definition of this type is system-dependent.
SLfloat32	A 32-bit floating-point type. The definition of this type is system-dependent.
SLfloat64	A 64-bit floating-point type. The definition of this type is system-dependent.
SLchar	A character type. All strings within the API, except where explicitly defined otherwise, are UTF-8, null-terminated, SLchar arrays. The definition of this type is system-dependent.
SLboolean	A Boolean type, where zero is false and all remaining values are true.
SLmillibel	A type for representing volume in millibels (mB), one thousandth of a Bel, one hundredth of a decibel.
SLmillisecond	A type for representing time in milliseconds (ms), one thousandth of a second).
SLmilliHertz	A type for representing frequency in milliHertz (mHz), one thousandth of a Hertz.
SLmillimeter	A type for representing distance in millimetres (mm), one thousandth of a meter.
SLmillidegree	A type for representing an angle in millidegrees (mdeg), one thousandth of a degree.
SLpermille	A type for representing a scale or factor in permille. One permille (1 ‰) is equal to a factor of 0.001. One thousand permille (1000 ‰) is equal to a factor of one.
SLmicrosecond	A type for representing time in microseconds, one millionth of a second).

Type	Description
`SLresult`	A type for standard OpenSL ES errors that all functions defined in the API return.

6 Functions

6.1 slCreateEngine Function

slCreateEngine

```
SL_API SLresult SLAPIENTRY slCreateEngine(
   SLObjectItf *pEngine,
   SLuint32 numOptions
   const SLEngineOption *pEngineOptions,
   SLuint32 numInterfaces,
   const SLInterfaceID *pInterfaceIds,
   const SLboolean * pInterfaceRequired
)
```

Description	Initializes the engine object and gives the user a handle.		
Pre-conditions	None.		
Parameters	pEngine	[out]	Pointer to the resulting engine object.
	numOptions	[in]	The number of elements in the options array. This parameter value is ignored if pEngineOptions is NULL. Similarly, a 0 value initializes the engine without the optional features being enabled.
	pEngineOptions	[in]	Array of optional configuration data. A NULL value initializes the engine without the optional features being enabled.
	numInterfaces	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	pInterfaceIds	[in]	An array of numInterfaces interface IDs, which the object should support. This parameter is ignored if numInterfaces is zero.
	pInterfaceRequired	[in]	An array of numInterfaces flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated and the error code SL_RESULT_FEATURE_UNSUPPORTED will be then returned. This parameter is ignored if numInterfaces is zero.
Return value	The return value can be one of the following:		

slCreateEngine	
	SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_FEATURE_UNSUPPORTED SL_RESULT_RESOURCE_ERROR SL_RESULT_ENGINEOPTION_UNSUPPORTED
Comments	If the requested version is not supported by the implementation, SL_RESULT_FEATURE_UNSUPPORTED is returned. The options supported by an individual implementation are implementation-dependent. Standardized options are documented in section 9.2.18. The engine is destroyed via the Destroy method in the `SLObjectItf` interface [see section 8.34]. The version of the implementation version returned can be queried using the QueryAPIVersion method in the `SLEngineCapabilitiesItf` interface [see section 8.18].
See Also	Engine object [see section 7.4].

6.2 slQueryNumSupportedEngineInterfaces Function

slQueryNumSupportedEngineInterfaces

```
SL_API SLresult SLAPIENTRY slQueryNumSupportedEngineInterfaces(
    SLuint32 * pNumSupportedInterfaces
);
```

Description	Queries the number of supported interfaces available on engine object.		
Parameters	pNumSupportedInterfaces	[out]	Identifies the number of supported interfaces available. Must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The number of supported interfaces will include both mandated and optional interfaces available for the engine object.		
See also	`slQuerySupportedEngineInterfaces()`, `SLEngineItf::QueryNumSupportedInterfaces` [see section 8.21].		

6.3 slQuerySupportedEngineInterfaces Function

slQuerySupportedEngineInterfaces			

`SL_API SLresult SLAPIENTRY slQuerySupportedEngineInterfaces(`
`    SLuint32 index,`
`    SLInterfaceID * pInterfaceId`
`);`			

Description	Queries the supported interfaces on engine object.		
Pre-conditions	None.		
Parameters	index	[in]	Index used to enumerate available interfaces. Supported index range is 0 to N-1, where N is the number of supported interfaces.
	pInterfaceId	[out]	Identifies the supported interface corresponding to the given index. Must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The number of supported interfaces will include both mandated and optional interfaces available for the engine object.		
See also	slQueryNumSupportedEngineInterfaces(), SLEngineItf::QueryNumSupportedInterfaces [see section 8.21].		

7 Object Definitions

This section documents all the object types supported by the API. Some object types are mandated to be supported only in a selection of the profiles. Where this is the case, the object's description will include a profile note stating this. If the object does not include a profile note, the object is mandated to be supported in all profiles.

Each object type has a list of ***Mandated Interfaces*** that must be supported for that object type. For each mandated object type in a given profile, an implementation must support the creation of at least one object with every mandated interface exposed on that object. Even if the object type itself is not mandated, if the implementation allows creation of objects of that type, it must still support all the mandated interfaces for the object type. The list of mandated interfaces may vary according to profile, as documented in the profiles notes. The mandated interface sections also document whether an interface is implicit or must be supported dynamically.

Besides of the mandated interfaces, an object is free to support any interfaces defined in this specification (and any vendor-specific interfaces). However, some interfaces specified in this specification make much more sense with a specific object type than certain other interfaces. Therefore, for information only, each object type has also a list of ***Applicable Optional interfaces***. The implementer is not limited to support only these listed interfaces, but these lists provide the application developer a hint concerning which optional interfaces might be supported.

7.1 3D Group

Description

In the majority of cases, 3D sound sources will be independently controlled and positioned and in these cases the application exposes the 3D interfaces on the player itself. However, there are circumstances where several sound sources may have the same 3D properties, including position. The 3D group object provides a convenient mechanism for grouping several sound sources with the same 3D properties. This is primarily used in order to conserve 3D resources but can also provide convenience to the application developer.

Typically, 3D resources can only be conserved when all players in the 3D group have the same data sink.

See section C.4 for an example using this object.

PROFILE NOTES
Creation of objects of this type is mandated only in the Game profile.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces (see section 3.1.6) to the object.

This interface is an implicit interface on this object.

SL3DLocationItf [see section 8.5]

This interface exposes controls for positioning and orienting the 3D group.

This interface is an implicit interface on this object.

SL3DDopplerItf [see section 8.3]

This interface exposes controls for the Doppler and velocity of the 3D group. This interface is a dynamic interface on this object. See section 3.1.6 for details about dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profile.

SL3DSourceItf [see section 8.8]

This interface exposes 3D source-oriented controls of the 3D group.

PROFILE NOTES
This interface is mandated only in the Game profile.

Applicable Optional Interfaces

SL3DMacroscopicItf [see section 8.7]

This interface exposes controls for setting the size (physical dimensions) of the 3D group.

SL3DHintItf [see section 8.5]

This interface defines the quality ‘hint’ of rendering applied to the 3D group.

7.2 Audio Player

Description

The audio player media object plays the piece of content specified by the data source performing any implicit decoding, applying any specified processing, and rendering it to the destination specified by the data sink.

See Appendix B: and Appendix C: for examples using this object.

Java Tone Sequence Playback

It is also possible to play back pre-created Java Tone Sequences using an Audio Player Object. To do this, the application needs to do the following:

1. Set the Address Data Locator Structure (`SLDataLocator_Address`) in the Audio Player's `SLDataSource` to contain a pointer to and the length of the byte array (array of `SLuint8`) containing the desired Java Tone Sequence.
2. Set the MIME Data Format Structure (`SLDataFormat_MIME`) in the `SLDataSource` to say that the format is "audio/x-tone-seq".

Please see JSR-135 [JSR135] ToneControl for details on the Java Tone Sequence bytecode format.

PROFILE NOTES
Java Tone Sequence playback is mandated only in the Phone and Game profiles. PCM sampled audio playback is mandated in all profiles.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used to add dynamic interfaces [see section 3.1.6] to the object.

This interface is an implicit interface on this object.

SLPlayItf [see section 8.37]

This interface controls the playback state of the audio player.

This interface is an implicit interface on this object.

SL3DDopplerItf [see section 8.3]

This interface exposes Doppler and velocity controls. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profile.

SL3DGroupingItf [see section 8.4]

This interface exposes controls for adding and removing a player to and from a 3D group.

PROFILE NOTES
This interface is mandated only in the Game profile.

SL3DLocationItf [see section 8.5]

This interface exposes controls for changing a player's location in 3D space.

PROFILE NOTES
This interface is mandated only in the Game profile.

SL3DSourceItf [see section 8.8]

This interface exposes player-specific 3D controls.

PROFILE NOTES
This interface is mandated only in the Game profile.

SLBufferQueueItf [see section 8.14]

This interface enables feeding data to the player using streaming buffers. Note that an attempt to instantiate an SLBufferQueueItf on a player whose data source is not of type SL_DATALOCATOR_BUFFERQUEUE will fail.

PROFILE NOTES
This interface is mandated only in the Game profile.

SLEffectSendItf [see section 8.17]

This interface controls a player's direct path and effect send levels.

PROFILE NOTES
This interface is mandated only in the Music and Game profiles.

SLMuteSoloItf [see section 8.33]

This interface exposes controls for selecting which of the player's channels are heard and silenced.

PROFILE NOTES
This interface is mandated only in the Game profile and only where the data source is not a Java Tone Sequence.

SLMetaDataExtractionItf [see section 8.26]

This interface exposes controls for metadata extraction. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Music and Game profiles.

SLMetaDataMessageItf [see section 8.26]

This interface exposes controls for metadata extraction during streaming and/or playback. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Music and Game profiles.

SLMetaDataTraversalItf [see section 8.28]

This interface exposes controls for metadata traversal. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Music and Game profiles.

SLPrefetchStatusItf [see section 8.39]

This interface controls the prefetch state of the audio player.

SLRatePitchItf [see section 8.41]

The interface controls the rate and pitch in the audio player. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profile and only where the data source is not a Java Tone Sequence.

SLSeekItf [see section 8.43]

This interface controls the position of the playback head and any looping of playback.

PROFILE NOTES

The SetLoop is mandated for end-to-end looping in the phone profile and mandated for looping over an arbitrary loop region in the Music and Game profiles.

SLVolumeItf [see section 8.48]

This interface exposes volume-related controls.

PROFILE NOTES

The EnableStereoPosition(), IsEnabledStereoPosition(), SetStereoPosition() and GetStereoPosition() methods are mandated only in the Music and Game profiles.

Applicable Optional Interfaces

SL3DMacroscopicItf [see section 8.7]

This interface exposes controls for setting the size (physical dimensions) of the player (3D sound source).

SL3DHintItf [see section 8.5]

This interface defines the quality 'hint' of rendering applied to the 3D audio source.

SLBassBoostItf [see section 8.13]

This interface controls a player-specific bass boost effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLDynamicSourceItf [see section 8.18]

This interface is deprecated. Use SLDynamicSourceSinkChangeItf [see section 8.19] instead.

This interface enables changing the data source of the player post-creation.

SLDynamicSourceSinkChangeItf [see section 8.19]

This interface enables changing the data source or the data sink of the object post-creation.

SLEnvironmentalReverbItf [see section 8.23]

This interface controls a player-specific reverb effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLEqualizerItf [see section 8.24]

This interface controls a player-specific equalizer effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLPitchItf [see section 8.36]

This interface controls the pitch shifting without changing the playback rate. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLPresetReverbItf [see section 8.40]

This interface controls a player-specific reverb effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLPlaybackRateItf [see section 8.38]

This interface exposes playback rate related controls. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLVirtualizerItf [see section 8.46]

This interface exposes controls over a player-specific virtualization effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLVisualizationItf [see section 8.47]

This interface provides data for visualization purposes.

7.3 Audio Recorder

Description

The audio recorder media object records the piece of content to the destination specified by the data sink capturing it from the input specified by the data source and performing any specified encoding or processing.

See section B.1.2 for an example using this object.

PROFILE NOTES
This object is a standardized extension and consequently optional in all profiles.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces [see section 3.1.6] to the object.

This interface is an implicit interface on this object.

SLRecordItf [see section 8.42]

This interface controls the recording state of the audio player.

This interface is an implicit interface on this object.

SLAudioEncoderItf [see section 8.10]

This interface exposes audio encoder functionality.

SLAudioEncoderCapabilitiesItf [see section 8.11]

This interface exposes methods for querying audio encoder capabilities.

Applicable Optional Interfaces

SLBassBoostItf [see section 8.13]

This interface controls the bass boost effect of the recorder. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLBufferQueueItf [see section 8.14]

This interface enables accepting data from a recorder using streaming buffers. Note that an attempt to instantiate an `SLBufferQueueItf` on a recorder whose data sink is not of type `SL_DATALOCATOR_BUFFERQUEUE` will fail.

SLDynamicSourceItf [see section 8.18]

This interface is deprecated. Use SLDynamicSourceSinkChangeItf [see section 8.19] instead.

This interface enables changing the data source of the recorder post-creation.

SLDynamicSourceSinkChangeItf [see section 8.19]

This interface enables changing the data source or the data sink of the object post-creation.

SLEqualizerItf [see section 8.24]

This interface controls the equalizer effect of the recorder. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLVisualizationItf [see section 8.47]

This interface provides data for visualization purposes.

SLVolumeItf [see section 8.48]

This interface exposes volume-related controls.

7.4 Engine Object

Description

This object type is the entry point of the API. An implementation shall enable creation of at least one such object, but attempting to create more instances (either by a single application or by several different applications) may fail.

The engine object supports creation of all the API's objects via its `SLEngineItf` interface, and querying of the implementation's capabilities via its `SLEngineCapabilitiesItf` interface.

See Appendix B: and Appendix C: for examples using this object.

Creation

An engine object is created using the global function `slCreateEngine()` [see section 6.1].

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces (see section 3.1.6) to the object.

This interface is an implicit interface on this object.

SLEngineItf [see section 8.21]

This interface exposes methods for creation of all the API's objects.

This interface is an implicit interface on this object.

SLEngineCapabilitiesItf [see section 8.22]

This interface enables querying current implementation capabilities.

This interface is an implicit interface on this object.

SLThreadSyncItf [see section 8.44]

This interface enables synchronization of API callback and client application contexts.

This interface is an implicit interface on this object.

SLAudioIODeviceCapabilitiesItf [see section 8.9]

This interface exposes methods for querying available audio device capabilities.

This interface is an implicit interface on this object.

SLAudioDecoderCapabilitiesItf [see section 8.9]

This interface exposes methods for querying audio decoder capabilities.

SLAudioEncoderCapabilitiesItf [see section 8.11]

This interface exposes methods for querying audio encoder capabilities.

SL3DCommitItf [see section 8.1]

This interface exposes the global 3D commit control for all 3D parameters within an engine.

PROFILE NOTES
This interface is mandated only in the Game profile.

Applicable Optional Interfaces

SLDeviceVolumeItf [see section 0]

This interface controls audio input and output device-specific volumes.

7.5 LED Array I/O Device

Description

The LED array I/O device object encapsulates and controls a set of LEDs. Its functionality covers setting LED color, activating and deactivating LEDs.

PROFILE NOTES
This object is a standardized extension and consequently optional in all profiles.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLLEDArrayItf [see section 8.25]

This interface exposes all LED capabilities for a LED array `IODevice`.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces (see section 3.1.6) to the object.

This interface is an implicit interface on this object.

7.6 Listener Object

Description

The listener object is an abstract object that represents a listener to any sound sources positioned in 3D space. The listener does not have a data source or data sink and subsequently has no content associated directly with it.

An application can optionally create one or more listener objects. Non-3D sound sources are heard independent of the existence or non-existence of any listener. For 3D sound sources to be heard, the application must create at least one listener.

The listener typically has the same position and orientation as a camera in a 3D graphics scene.

Some implementations may support the creation of multiple listeners. The behavior in such cases is undefined.

See section B.5 and C.4 for examples using this object.

PROFILE NOTES
Creation of objects of this type is mandated only in the Game profile.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces (see section 3.1.6) to the object.

This interface is an implicit interface on this object.

SL3DDopplerItf [see section 8.3]

This interface exposes controls for the Doppler and velocity of the listener. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profile.

SL3DLocationItf [see section 8.5]

This interface exposes controls for positioning and orienting the listener.

PROFILE NOTES

This interface is mandated only in the Game profile.

7.7 Metadata Extractor Object

Description

This object can be used for reading metadata without allocating resources for media playback. Using this object is recommended particularly when the application is interested only in presenting metadata without playing the content and when it wants to present metadata of multiple files. The latter is useful for the generation of playlists for presentation purposes because an audio player would unnecessarily allocate playback resources.

PROFILE NOTES
Creation of objects of this type is mandated in Music and Game profiles.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces (see section 3.1.6) to the object.

This interface is an implicit interface on this object.

SLDynamicSourceItf [see section 8.18]

This interface is deprecated. Use SLDynamicSourceSinkChangeItf [see section 8.19] instead.

This interface exposes controls for changing the data source during the lifetime of the object, to be able to read metadata from multiple files without creating a new object for every single file.

This interface is an implicit interface on this object.

SLDynamicSourceSinkChangeItf [see section 8.19]

This interface enables changing the data source or the data sink of the object post-creation.

This interface is an implicit interface on this object.

SLMetaDataExtractionItf [see section 8.26]

This interface exposes controls for metadata extraction.

This interface is an implicit interface on this object.

SLMetaDataMessageItf [see section 8.27]

This interface exposes methods for setting metadata callbacks. To be used in conjunction with SLMetadataExtractionItf.

This interface is an implicit interface on this object.

SLMetaDataTraversalItf [see section 8.28]

This interface exposes controls for metadata traversal.

This interface is an implicit interface on this object.

7.8 MIDI Player Object

Description

The MIDI Player media object is used for all rendering of MIDI data. This includes both MIDI-based content files and MIDI-based wavetable instrument files. Further, individual MIDI messages not encapsulated within content files may be sent to a MIDI Player object via the optional MIDI Messages interface. Like the Audio Player media object, data sources for a MIDI Player generally include files and buffer queues and the primary data sinks are audio output devices. In addition, an optional data source (a Mobile DLS instrument bank file) and two additional optional data sinks (LED array and vibra I/O device) are also available.

See sections B.3, C.3 and C.4 for examples using this object.

PROFILE NOTES
Creation of objects of this type is mandated only in the Phone and Game profiles.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces (see section 3.1.6) to the object.

This interface is an implicit interface on this object.

SLPlayItf [see section 8.37]

This interface controls the playback state of the MIDI player. However, the play state does not affect controlling the MIDI Player with real-time MIDI messages via SLMIDIMessageItf; therefore, it is not necessary to use SLPlayItf if only real-time messages are used to control the MIDI player (with or without a soundbank source).

This interface is an implicit interface on this object.

SL3DDopplerItf [see section 8.3]

This interface exposes Doppler and velocity controls. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profile.

SL3DGroupingItf [see section 8.4]

This interface exposes controls for adding and removing a player to and from a 3D group.

PROFILE NOTES
This interface is mandated only in the Game profile.

SL3DLocationItf [see section 8.5]

This interface exposes controls for changing a player's location in 3D space.

PROFILE NOTES
This interface is mandated only in the Game profile.

SL3DSourceItf [see section 8.8]

This interface exposes player-specific 3D controls.

PROFILE NOTES
This interface is mandated only in the Game profile.

SLBufferQueueItf [see section 8.14]

This interface enables feeding data to the player using buffers. Note that an attempt to instantate an SLBufferQueueItf on a player whose data source is not of type `SL_DATALOCATOR_MIDIBUFFERQUEUE` will fail.

PROFILE NOTES
This interface is mandated only in the Game profile.

SLEffectSendItf [see section 8.17]

This interface controls a player's direct path and effect send levels.

PROFILE NOTES
This interface is mandated only in the Game profile.

SLMetaDataExtractionItf [see section 8.26]

This interface exposes controls for metadata extraction. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profile.

SLMetaDataMessageItf [see section 8.27]

This interface exposes controls for metadata extraction during streaming and/or playback. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profiles.

SLMetaDataTraversalItf [see section 8.28]

This interface exposes controls for metadata traversal. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profile.

SLMIDIMessageItf [see section 8.29]

The `SLMIDIMessageItf` interface exposes methods to send messages to a MIDI-based player and establish callbacks to get MIDI information in runtime.

PROFILE NOTES
This interface is mandated only in the Phone and Game profiles.

SLMIDITimeItf [see section 8.32]

The `SLMIDITimeItf` interface exposes methods to determine duration, to set and get position, and to set and get loop points in MIDI ticks.

PROFILE NOTES
This interface is mandated only in the Phone and Game profiles.

The SetLoopPoints() and GetLoopPoints() methods are mandated only in the Game profile.

SLMIDITempoItf [see section 8.31]

The `SLMIDITempoItf` interface exposes methods to set and get information about a MIDI-based player's tempo.

PROFILE NOTES
This interface is mandated only in the Phone and Game profiles.

SLMIDIMuteSoloItf [see section 8.30]

The `SLMIDIMuteSoloItf` interface exposes methods to mute and solo MIDI channels and tracks, and to get the number of tracks.

PROFILE NOTES
This interface is mandated only in the Game profiles.

SLPrefetchStatusItf [see section 8.39]

This interface controls the prefetch state of the MIDI player.

PROFILE NOTES
This interface is mandated only in the Phone and Game profiles.

SLSeekItf [see section 8.43]

This interface controls the position of the playback head and any looping of playback.

PROFILE NOTES
The SetLoop is mandated for end-to-end looping in the phone profile and mandated for looping over an arbitrary loop region in the Music and Game profiles.

SLVolumeItf [see section 8.48]

This interface exposes volume-related controls.

PROFILE NOTES
The EnableStereoPosition(), IsEnabledStereoPosition(), SetStereoPosition() and GetStereoPosition() methods are mandated only in the Music and Game profile.

Applicable Optional Interfaces

SL3DMacroscopicItf [see section 8.7]

This interface exposes controls for setting the size (physical dimensions) of the player (3D sound source).

SL3DHintItf [see section 8.5]

This interface defines the quality ‘hint’ of rendering applied to the 3D audio source.

SLBassBoostItf [see section 8.13]

This interface controls a player-specific bass boost effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLDynamicSourceItf [see section 8.18]

This interface is deprecated. Use SLDynamicSourceSinkChangeItf [see section 8.19] instead.

This interface enables changing the data source of the player post-creation.

SLDynamicSourceSinkChangeItf [see section 8.19]

This interface enables changing the data source or the data sink of the object post-creation.

SLEnvironmentalReverbItf [see section 8.23]

This interface controls a player-specific reverb effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLEqualizerItf [see section 8.24]

This interface controls a player-specific equalizer effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLPitchItf [see section 8.36]

This interface control the pitch shifting without changing the playback rate. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLPresetReverbItf [see section 8.40]

This interface controls a player-specific reverb effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLPlaybackRateItf [see section 8.38]

This interface exposes playback rate related controls. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLVirtualizerItf [see section 8.46]

This interface exposes controls over a player-specific virtualization effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLVisualizationItf [see section 8.47]

This interface provides data for visualization purposes.

7.9 Output Mix

Description

The output mix object represents a set of audio output devices to which one audio output stream is sent. The application retrieves an output mix object from the engine and may specify that output mix as the sink for a media object. The engine must support at least one output mix, though it may support more. The API does not provide a direct audio output IO-device as a sink for media objects.

An output mix is a logical object; it does not (necessarily) represent a physical mix. Thus the actual implementation of the mixing defined logically by the mix objects and their association with media objects is an implementation detail. The output mix does not represent the system's main mix. Furthermore, a mix object represents the application's contribution to the output; the implementation may mix this contribution with output from other sources.

The engine populates the output mix with the default set of audio output devices. The application may request rerouting of that mix via calls to add and remove devices, but whether those requests are fulfilled is entirely the prerogative of the implementation. Furthermore, the implementation may perform its own rerouting of the output mix. In this case, the implementation makes the application aware of changes to the output mix via a notification.

Manipulation of the output mixes leverages the use of device IDs to specify the device(s) operated on. The engine includes a special ID, called the *default device ID*, which represents a set of one or more devices to which the implementation deems audio output should go by default. Although the application may use the default device ID when manipulating an output mix, only the implementation may alter the physical devices this ID represents. Furthermore, the implementation may change the mapping to physical devices dynamically.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces (see section 3.1.6) to the object.

This interface is an implicit interface on this object.

SLOutputMixItf [see section 8.35]

This interface exposes controls for querying the associated destination output devices.

This interface is an implicit interface on this object.

SLEnvironmentalReverbItf [see section 8.23]

This interface exposes controls for an environmental reverb. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Game profile.

SLEqualizerItf [see section 8.24]

This interface exposes controls over an equalizer effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Music and Game profiles.

SLPresetReverbItf [see section 8.40]

This interface exposes preset controllable reverb. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Music profile.

SLVirtualizerItf [see section 8.46]

This interface exposes controls over a virtualization effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

PROFILE NOTES
This interface is mandated only in the Music and Game profiles.

SLVolumeItf [see section 8.48]

This interface exposes volume-related controls.

PROFILE NOTES
The EnableStereoPosition(), IsEnabledStereoPosition(), SetStereoPosition() and GetStereoPosition() methods are mandated only in the Game and Music profiles.

Applicable Optional Interfaces

SLBassBoostItf [see section 8.13]

This interface controls the bass boost effect. This interface is a dynamic interface on this object. See section 3.1.6 for details on dynamic interfaces.

SLVisualizationItf [see section 8.47]

This interface provides data for visualization purposes.

7.10 Vibra I/O Device

Description

The Vibra I/O device object controls device vibration. Its functionality is limited to activate / deactivate the vibration function of the device, as well as setting its frequency and intensity, if supported.

PROFILE NOTES
This object is a standardized extension and consequently optional in all profiles.

Mandated Interfaces

SLObjectItf [see section 8.34]

This interface exposes basic object functionality.

This interface is an implicit interface on this object.

SLDynamicInterfaceManagementItf [see section 8.17]

This interface is used for adding dynamic interfaces (see section 3.1.6) to the object.

This interface is an implicit interface on this object.

SLVibraItf [see section 8.45]

This interface exposes all vibration functionality for a Vibra I/O Device.

This interface is an implicit interface on this object.

8 Interface Definitions

This section documents all the interfaces and methods in the API.

Almost all methods generate result codes, whether synchronously or asynchronously. Such methods must return either one of the explicit result codes listed in the method's documentation or one of the following result codes:

- `SL_RESULT_RESOURCE_ERROR`
- `SL_RESULT_RESOURCE_LOST`
- `SL_RESULT_INTERNAL_ERROR`
- `SL_RESULT_UNKNOWN_ERROR`
- `SL_RESULT_OPERATION_ABORTED`
- `SL_RESULT_FEATURE_UNSUPPORTED`

For a full definition of these result codes see section 9.2.46.

8.1 SL_IID_NULL

SL_IID_NULL is a placeholder interface ID for the interface array used when creating an object. SL_RESULT_FEATURE_UNSUPPORTED will be returned when the engine tries to create an object that has SL_IID_NULL as a required interface. Also, using it in SLObject::GetInterface() or SLDynamicInterfaceManagement::AddInterface() will always fail.

Interface ID

2cc1cd80-e5e1-4432-a3f4-4657e6795210

8.2 SL3DCommitItf

Description

By default, all interfaces *commit* their settings to the signal processing layer immediately. This can result in unnecessary recalculations of 3D parameters and does not allow the developer to set up the 3D scene in one atomic operation. This interface exposes commit controls for controlling when changes to 3D interfaces are sent to the signal-processing layer.

This interface controls when changes to the following interfaces are sent to the signal processing system:

- `SL3DLocationItf`
- `SL3DDopplerItf`
- `SL3DSourceItf`
- `SL3DMacroscopicItf`

This affects all objects that can expose these interfaces, namely: listeners, players and 3D groups. All other interfaces, including the `SL3DGroupingItf` interface, are committed to the signal processing system immediately, regardless of this interface's settings.

Applications are advised to defer 3D settings when possible, as this will reduce the amount of unnecessary internal parameter calculations.

When in deferred mode, all "get" methods for the 3D interfaces will return the latest "set" values, even if they have not yet been committed.

This interface is supported on the engine object [see section 7.4].

See section B.5.2 for an example using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_3DCOMMIT;

struct SL3DCommitItf_;
typedef const struct SL3DCommitItf_* const * SL3DCommitItf;

struct SL3DCommitItf_ {
   SLresult (*Commit) (
         SL3DCommitItf self
   );
```

```
      SLresult (*SetDeferred) (
            SL3DCommitItf self,
            SLboolean deferred
      );
   };
```

Interface ID

3564ad80-dd0f-11db-9e19-0002a5d5c51b

Defaults

Commit mode: Immediate (not deferred).

Methods

Commit			

`SLresult (*Commit) (`
`   SL3DCommitItf self`
`);`			

Description	Commits all changes to all 3D interfaces, except `3DGrouping`.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS		
Comments	It is legal to call this method when the commit mode is immediate (that is, not deferred); it will have no effect.		

SetDeferred			
`SLresult (*SetDeferred) ( SL3DCommitItf self, SLboolean deferred );`			
Description	Enables or disables deferred committing of 3D parameters.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`deferred`	[in]	If true, all 3D parameter changes will be deferred until `Commit()` is called. If false, all parameter deferring is disabled.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS		
Comments	Parameters are not automatically committed when transferring from deferred to immediate mode. Developers should call `Commit()` before returning to immediate mode to ensure no parameter changes are lost.		

8.3 SL3DDopplerItf

Description

This interface controls the Doppler of a listener, player, or 3D group. If this interface is exposed on a 3D group, the methods control the 3D characteristics of all the players in the 3D group.

The following restrictions must be adhered to when exposing this interface on a player object:

- This interface can be exposed on a player only if the `SL3DLocationItf` interface is exposed on the same player. This interface can be exposed on a player at creation as long as the `SL3DLocationItf` is also exposed at creation.

Parameter changes made using this interface may be deferred using the `SL3DCommitItf` interface [see section 8.1].

If the Doppler interface is exposed on a player or 3D Group (3D source), the players involved will have the Doppler effect applied (unless the Doppler factor is zero), regardless of whether the listener has exposed the Doppler interface. If the Doppler interface is not exposed on a 3D source, the 3D source will not have the Doppler effect applied to it, regardless of whether the listener has exposed the Doppler interface.

This interface is supported on the Audio Player [see section 7.2], MIDI Player [see section 7.8], 3D Group [see section 7.8] and Listener [see section 7.6] objects.

See sections B.5.2 and C.4 for examples using this interface.

Doppler

When a 3D source is moving at a speed relative to the listener, its perceived pitch changes. This change is called *Doppler shift* and is most noticeable if a 3D source moves quickly close to the listener. To calculate the amount of pitch shift in a real world situation, it is only necessary to know relative velocities of the source and listener, in addition to knowledge of the physical medium through which the sound waves travel (which is usually air). These parameters can in turn be calculated from knowledge of the spatial positions of the sound source and listener as they vary with time.

However, OpenSL ES does not calculate Doppler shifts from the 3D source and listener locations because it is usually more convenient for the developer to directly specify the *velocities* of the 3D source and listener. This allows the developer more flexibility, as these velocities can be decoupled from the actual physical velocities of the 3D source and listener. If desired, the developer can easily calculate the 3D source and listener velocities that correspond directly to their physical movement, thus linking the two again.

It is good practice to use Doppler only on selected 3D sources on which the effect is going to be most effective or noticeable, since Doppler processing can use additional memory and processing power. For example, Doppler works well for speeding bullets but not for walking characters. Developers can conserve resources by exposing this interface only on 3D sources that require the Doppler effect.

Doppler can be simulated as a change in pitch. The following is the recommended algorithm for calculating a pitch multiplier for a given Doppler:

$$d = \begin{cases} \dfrac{(\mathbf{l}_p)}{|\mathbf{l}_p|} & \text{if player is head relative} \\[2ex] \dfrac{(\mathbf{l}_p - \mathbf{l}_l)}{|\mathbf{l}_p - \mathbf{l}_l|} & \text{otherwise} \end{cases}$$

$$s_l = \mathbf{v}_l \cdot d$$

$$s_p = \mathbf{v}_p \cdot d$$

$$D = \frac{D_l \times D_p}{10^6}$$

$$p = \begin{cases} 1, & \text{if } D = 0 \\ \max\left(0, \dfrac{c + D \times s_l}{c + D \times s_p}\right), & \text{otherwise} \end{cases}$$

where:

p is the pitch multiplier due to Doppler.

$A \cdot B$ is the dot product of the vectors A and B.

$\mathbf{v}_l$ is the listener's velocity vector.

$\mathbf{v}_p$ is the 3D source's velocity vector.

$\mathbf{l}_l$ is the listener's location vector.

$\mathbf{l}_p$ is the 3D source's location vector.

D_l is the listener's Doppler factor.

D_p is the 3D source's Doppler factor.

c is the speed of sound (about 340000 mm/s).

The listener’s velocity and Doppler factor are used for calculating the 3D source’s Doppler. If the SL3DDopplerItf interface is not exposed on the listener, the default listener velocity and Doppler factor values are used in the calculations.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_3DDOPPLER;

struct SL3DDopplerItf_;
typedef const struct SL3DDopplerItf_ * const * SL3DDopplerItf;

struct SL3DDopplerItf_ {
   SLresult (*SetVelocityCartesian) (
         SL3DDopplerItf self,
         const SLVec3D *pVelocity
   );
   SLresult (*SetVelocitySpherical) (
         SL3DDopplerItf self,
         SLmillidegree azimuth,
         SLmillidegree elevation,
         SLmillimeter speed
   );
   SLresult (*GetVelocityCartesian) (
         SL3DDopplerItf self,
         SLVec3D *pVelocity
   );
   SLresult (*SetDopplerFactor) (
         SL3DDopplerItf self,
         SLpermille dopplerFactor
   );
   SLresult (*GetDopplerFactor) (
         SL3DDopplerItf self,
         SLpermille *pDopplerFactor
   );
};
```

Interface ID

b45c9a80-ddd2-11db-b028-0002a5d5c51b

Defaults

Velocity (x, y, z): (0 mm/s, 0 mm/s, 0 mm/s)

Doppler factor: 1000 ‰

Methods

SetVelocityCartesian			
`SLresult (*SetVelocityCartesian) ( SL3DDopplerItf self, const SLVec3D *pVelocity );`			
Description	Sets the object's velocity.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pVelocity`	[in]	Pointer to a vector containing the velocity in right-handed Cartesian coordinates. The velocities are expressed in millimeters per second.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The velocity is only used in the Doppler calculations. It does not effect the object's location.		

SetVelocitySpherical

```
SLresult (*SetVelocitySpherical) (
    SL3DDopplerItf self,
    SLmillidegree azimuth,
    SLmillidegree elevation,
    SLmillimeter speed
);
```

Description	Sets the object’s velocity using spherical coordinates.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	azimuth	[in]	The azimuth angle in millidegrees. The valid range is [-360000, 360000].
	elevation	[in]	The elevation angle in millidegrees. The valid range is [-90000, 90000].
	speed	[in]	The speed in millimeters per second. The valid range is [0, SL_MILLIMETER_MAX].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The velocity is only used in the Doppler calculations. It does not effect the object’s location. See the Location section in documentation for SL3DLocationItf [see section 8.5] for a definition of azimuth and elevation.		

GetVelocityCartesian			
`SLresult (*GetVelocityCartesian) ( SL3DDopplerItf self, SLVec3D *pVelocity );`			
Description	Gets the object's velocity.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pVelocity	[out]	Pointer to a vector to receive the velocity in right-handed Cartesian coordinates. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The accuracy of the output velocity is limited: one or more of the components (x, y or z) can differ by (a) up to 4% of the largest (absolute) vector component, or (b) 1 mm from the accurate value, which ever is greater.		

SetDopplerFactor

```
SLresult (*SetDopplerFactor) (
    SL3DDopplerItf self,
    SLpermille dopplerFactor
);
```

Description	Sets the object's Doppler factor.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	dopplerFactor	[in]	Doppler factor in permille. A value of zero disables the Doppler effect. The valid range is [0, 10000].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	A 3D source's Doppler factor is multiplied by the listener's Doppler factor to determine the final Doppler factor applied to the sound source. Consequently, if the listener's Doppler factor is zero, all Doppler effects are disabled. If the SL3DDopplerItf interface is not exposed on the listener, the listener's default Doppler factor (1000 ‰) is used in this calculation. If a Doppler effect is never required on the 3D source, the developer is advised not to expose this interface, as this will save resources.		

GetDopplerFactor

```
SLresult (*GetDopplerFactor) (
    SL3DDopplerItf self,
    SLpermille *pDopplerFactor
);
```

Description	Gets the object's Doppler factor.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pDopplerFactor	[out]	Pointer to a location to receive the current Doppler factor. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.4 SL3DGroupingItf

Description

This interface sets the player's 3D group. A player can be added to a 3D group, removed from one and moved between 3D groups. When this interface is exposed on a player, it must be added to a 3D group (using `Set3DGroup()`) in order for the player to be heard.

The following restrictions must be adhered to when exposing this interface on a player:

- This interface can only be exposed when creating a player object. It cannot be dynamically added using the `SLDynamicInterfaceManagementItf` interface [see section 8.17].
- This interface is mutually exclusive with the `SL3DLocationItf` interface; it is not possible to expose this interface at the same time as the `SL3DLocationItf` interface.

Exposing this interface on a player renders the player in 3D.

This interface is supported on the Audio Player [see section 7.2] and MIDI Player [see section 7.8] objects.

See section C.4 for an example using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_3DGROUPING;


struct SL3DGroupingItf ;
typedef const struct SL3DGroupingItf _ * const * SL3DGroupingItf;

struct SL3DGroupingItf_ {
   SLresult (*Set3DGroup) (
         SL3DGroupingItf self,
         SLObjectItf group
   );
   SLresult (*Get3DGroup) (
         SL3DGroupingItf self,
         SLObjectItf *pGroup
   );
};
```

Interface ID

ebe844e0-ddd2-11db-b510-0002a5d5c51b

Defaults

3D group: `NULL` (that is, no 3D group)

Methods

Set3DGroup			

`SLresult (*Set3DGroup) (`
`   SL3DGroupingItf self,`
`   SLObjectItf group`
`);`			

Description	Sets the 3D group for the player, removing the player from any previous 3D group.		
Pre-conditions	The 3D group being set must be in the realized state.		
Parameters	`self`	[in]	Interface self-reference.
	`group`	[in]	The 3D group to add the player to. If `group` is equal to `NULL`, the player is no longer in any 3D group. The 3D group must be in the realized state.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_MEMORY_FAILURE`		

Comments	When a player is no longer a member of a 3D group, it has no 3D information and so is not heard until the player is added back to 3D group.		

Get3DGroup			
`SLresult (*Get3DGroup) ( SL3DGroupingItf self, SLObjectItf *pGroup );`			
Description	Gets the 3D group for the player.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pGroup	[out]	Pointer to location to receive the 3D group of which the player is a member. If the player is not a member of 3D group, this will return NULL. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.5 SL3DHintItf

Description

SL3DHintItf defines the quality 'hint' of rendering applied to the associated 3D source. If this interface is exposed on a 3D group, the methods control the rendering quality of all sources in the 3D group. The hint value relates to the importance of correctly positioning the 3D source. The higher the qualityHint value, the greater importance to correctly positioning the 3D source the implementation should give.

The handling 3D hint is highly implementation specific, and may be based on many factors. Accordingly the interpretation and audible result of a hint may vary in response to any change to the audio state, as well as the degree to which the implementation takes into account hinting during processing.

The following restriction must be adhered to when exposing this interface on a player object:

- This interface can be exposed on a player only if the SL3DLocationItf interface is exposed on the same player. This interface can be exposed on a player at creation as long as SL3DLocationItf is also exposed at creation.

Parameter changes made using this interface may be deferred using the SL3DCommitItf interface [see section 8.1].

This interface is optionally supported on the Audio Player [see section 7.2], MIDI Player [see section 7.8] and 3D Group [see section 7.8] objects.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_3DHINT;

struct SL3DHintItf_;
typedef const struct SL3DHintItf_ * const * SL3DHintItf;

struct SL3DHintItf_ {
   SLresult (*SetRenderHint) (
         SL3DHintItf self,
         SLuint16 qualityHint
   );
   SLresult (*GetRenderHint) (
         SL3DHintItf self,
         SLuint16 *pQualityHint
   );
};
```

Interface ID

4a914bb0-c5db-11df-bd3b-0800200c9a66

Defaults

Render Type: **SL_3DHINT_OFF – 0x0000**

Methods

SetRenderHint

```
   SLresult (*SetRenderHint) (
         SL3DHintItf self,
         SLuint16 qualityHint
   );
```

Description	Sets the 3D source’s rendering quality hint.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference
	qualityHint	[in]	The requested rendering quality. See the **SL_3DHINT** macros for more information.

SetRenderHint			

Return value	The return value can be one of the following:
	
	`SL_RESULT_SUCCESS`
	
	`SL_RESULT_PARAMETER_INVALID`		

GetRenderHint			
`SLresult (*GetRenderHint) ( SL3DHintItf self, SLuint16 *pQualityHint );`			
Description	Retrieves the 3D source's rendering quality as presently in use by the implementation.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference
	`pQualityHint`	[out]	The current rendering quality. See the **`SL_3DHINT`** macros for more information.

Return value	The return value can be one of the following:
	
	`SL_RESULT_SUCCESS`
	
	`SL_RESULT_PARAMETER_INVALID`		

8.6 SL3DLocationItf

Description

This interface controls the location and orientation in 3D space of a listener, player, or 3D group. If this interface is exposed on a 3D group, the methods control the 3D characteristics of all the players in the 3D group.

The following restrictions must be adhered to when exposing this interface on a player object:

- This interface can be exposed only when creating a player object. It cannot be dynamically added using the `SLDynamicInterfaceManagementItf` interface [see section 8.17].
- This interface is mutually exclusive with the `SL3DGroupingItf` interface; it is not possible to expose this interface at the same time as the `SL3DGroupingItf` interface.

Exposing this interface on a player object causes the player to be rendered in 3D.

Parameter changes made using this interface may be deferred using the `SL3DCommitItf` interface [see section 8.1].

This interface is supported on the Audio Player [see section 7.2], MIDI Player [see section 7.8], 3D Group [see section 7.8] and Listener [see section 7.6] objects.

See section B.5 and C.4 for examples using this interface.

Location

The location of an object can be specified using either Cartesian or spherical coordinates. All coordinates are specified using either world coordinates or listener-relative coordinates (if the 3D source is in head relative mode [see section 8.8]).

A location is specified in Cartesian coordinates using a vector that specifies the distance x, y, and z-axis position, as shown in the following left diagram. A location is specified using spherical coordinates by specifying the distance from the reference point, and the azimuth and elevation angles, as shown in the following right diagram.

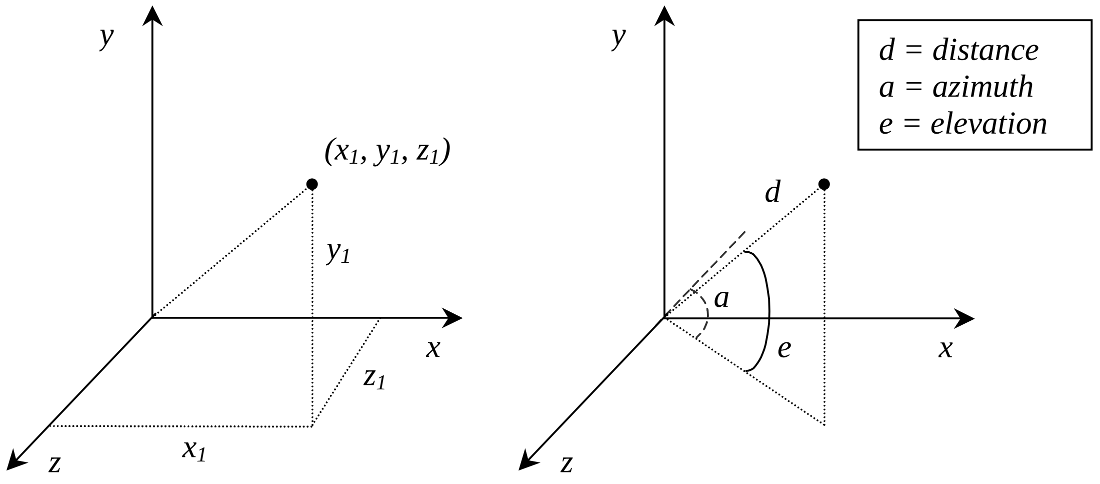


Figure 24: Specifying object location

The conversion from spherical coordinates to Cartesian coordinates is defined by the following equations:

$$x = distance \times \cos(elevation) \times \sin(azimuth)$$
$$y = distance \times \sin(elevation)$$
$$z = distance \times \cos(elevation) \times -\cos(azimuth)$$

In addition to the above mentioned mechanisms, which are setter methods for specifying the location relative to the origin (or to the listener when in the head relative mode, see `SetHeadRelative()` method in section 8.8), there is a third mechanism, namely the Move method, to specify the location relative to the previous location using Cartesian coordinates.

Orientation

Many sound sources are omni-directional, that is, they radiate sound equally in all directions, so that the energy they emit is the same regardless of their orientation. However, other sound sources radiate more energy in one direction than others. OpenSL ES allows the application to model this effect by specifying the orientation of the 3D sources and listener (as described here) and a 3D source's sound cone [see section 8.8].

The orientation of an object can be specified using three alternative methods:

- `SetOrientationAngles()`, for setting it using three rotation angles relative to the default orientation.
- `SetOrientationVectors()`, for setting it using orientation vectors relative to the default orientation.

- `Rotate()`, for setting it using a rotation axis and a rotation angle relative to the current orientation.

Orientation Angles

An orientation is expressed using rotation angles by specifying the heading, pitch and roll rotations about the coordinate axes of the object. The new orientation is specified relative to the initial orientation. Positive rotation directions around the coordinate axes are counterclockwise when looking towards the origin from a positive coordinate position on each axis. The initial orientation is facing out towards the negative z-axis of the world, up-direction being towards the positive y-axis of the world. The heading specifies the rotation about the object's y-axis, the pitch specifies the heading about the object's x-axis and the roll specifies the rotation about the object's z-axis. The rotation is applied in the order: heading, pitch, roll. Since the rotation angles are defined to be about the axes of the object, not of the world, the consequence is that the heading rotation affects both the pitch and roll and the pitch rotation affects the roll.

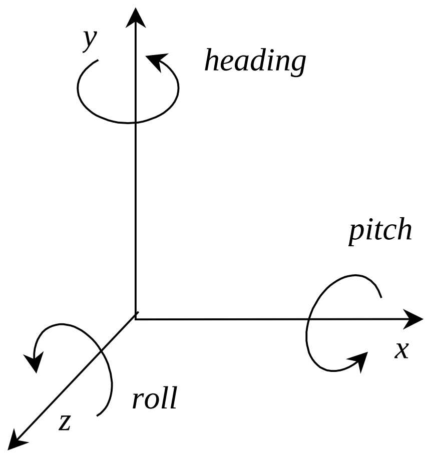


Figure 25: Orientation angles

In the case of the listener, orientation angles map nicely to the physical movements of the head: heading => turning around, pitch => nodding and roll => tilting the head left and right.

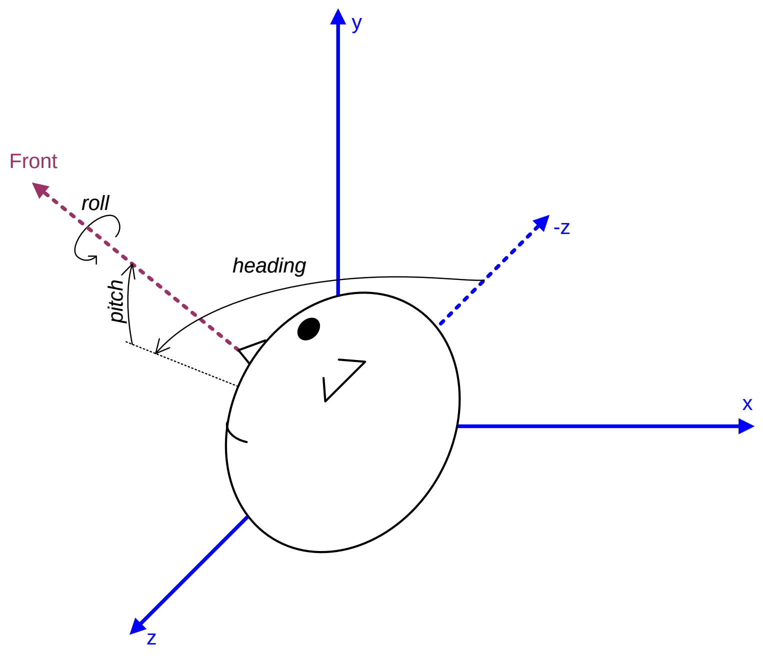


Figure 26: Orientation angles(2)

The conversion (without scaling) from heading, pitch and roll to Front and Up vectors is defined by the following equations:

$$x_{FRONT} = -\sin(heading) \times \cos(pitch)$$
$$y_{\mathrm{FRONT}} = \sin(pitch)$$
$$\mathrm{z}_{\mathrm{FRONT}} = -\cos(heading) \times \cos(pitch)$$
$$x_{\mathrm{UP}} = -\sin(roll) \times \cos(heading) + \cos(roll) \times \sin(pitch) \times \sin(heading)$$
$$y_{\mathrm{UP}} = \cos(pitch) \times \cos(roll)$$
$$\mathrm{z}_{\mathrm{UP}} = \sin(roll) \times \sin(heading) + \cos(roll) \times \cos(heading) \times \sin(pitch)$$

Orientation Vectors

When specifying the orientation, we consider the following vectors:

- *Front*: this vector specifies the frontal direction of the object.
- *Up*: this vector specifies the upward direction of the object. The *Up* vector is perpendicular to the *Front* vector.
- *Above*: this vector specifies a direction above the object, on the plane defined by the *Front* and *Up* vectors.

These vectors are shown for the listener's orientation in the diagram below.

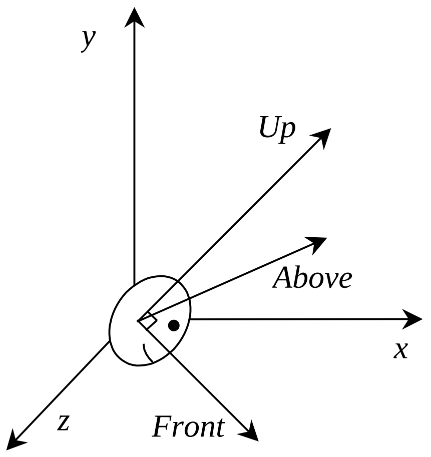


Figure 27: Orientation vectors

The *Right* and *Up* vectors of the object are calculated by first calculating the *Right* vector as the cross product of the *Front* vector and the *Above* vector, and then the *Up* vector as a cross product of the *Right* and *Front* vectors.

The method `SetOrientation()` vector has Front and Above vectors as parameters, but `GetOrientation()` returns the Up vector. The benefit of using the Above vector instead of the Up vector in the setter is that the application does not need to calculate the Up vector. The difference between the Above and Up vectors is that the Up vector must have a 90 degree angle to the Front vector, but the Above vector does not need (but is allowed) to have it. For example, if the application never wants to roll the listener’s head and the listener is just turning around and maybe nodding a little, the application can use a constant Above vector (0, 1000, 0) and just calculate the Front vector.

Rotate Method

The Rotate method can be used in order to turn the object from its current orientation by defining the rotation axis and the amount of rotation using a theta angle. This rotation follows the right-hand rule, so if the rotation axis points toward the user, the rotation will be counterclockwise. The following diagram illustrates altering the existing orientation using the Rotate method.

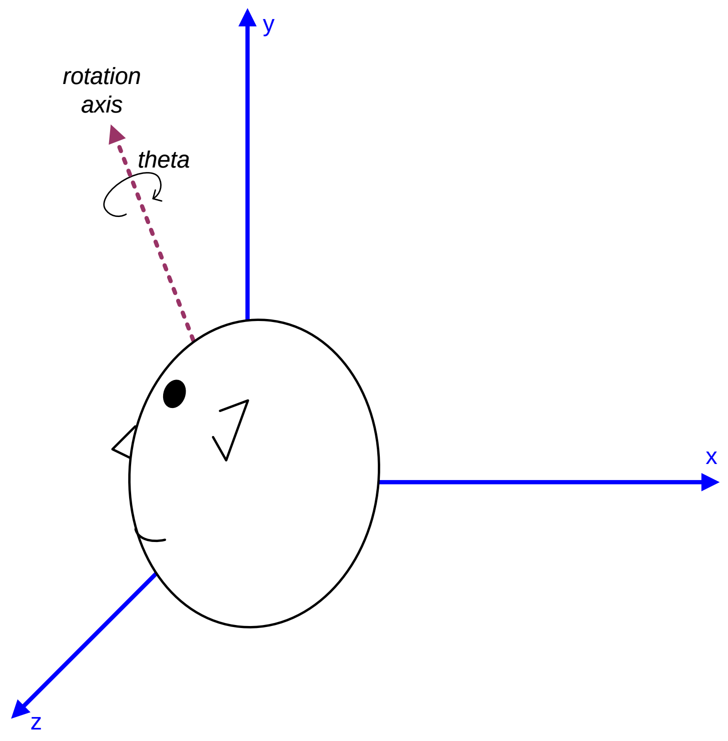


Figure 28: Rotate method

Prototype

```
SL_API extern const SLInterfaceID SL_IID_3DLOCATION;

struct SL3DLocationItf_;
typedef const struct SL3DLocationItf_ * const * SL3DLocationItf;

struct SL3DLocationItf_ {
   SLresult (*SetLocationCartesian) (
         SL3DLocationItf self,
         const SLVec3D *pLocation
   );
   SLresult (*SetLocationSpherical) (
         SL3DLocationItf self,
         SLmillidegree azimuth,
         SLmillidegree elevation,
         SLmillimeter distance
   );
   SLresult (*Move) (
         SL3DLocationItf self,
         const SLVec3D *pMovement
   );
   SLresult (*GetLocationCartesian) (
         SL3DLocationItf self,
         SLVec3D *pLocation
   );
   SLresult (*SetOrientationVectors) (
         SL3DLocationItf self,
         const SLVec3D *pFront,
         const SLVec3D *pAbove
   );
   SLresult (*SetOrientationAngles) (
         SL3DLocationItf self,
         SLmillidegree heading,
         SLmillidegree pitch,
         SLmillidegree roll
   );
   SLresult (*Rotate) (
         SL3DLocationItf self,
         SLmillidegree theta,
         const SLVec3D *pAxis
   );
   SLresult (*GetOrientationVectors) (
         SL3DLocationItf self,
         SLVec3D *pFront,
         SLVec3D *pUp
   );
};
```

Interface ID

2b878020-ddd3-11db-8a01-0002a5d5c51b

Defaults

Location (x, y, z): (0 mm, 0 mm, 0 mm)

Front orientation (x, y, z): (0, 0, -1000)

Up orientation (x, y, z): (0, 1000, 0)

That is, the initial position is the origin and the initial orientation is towards the negative Z-axis, the up-direction being towards the positive Y-axis. In rotation angles this equals: heading = 0 degrees; pitch = 0 degrees and roll = 0 degrees.

Methods

SetLocationCartesian

```
SLresult (*SetLocationCartesian) (
   SL3DLocationItf self,
   const SLVec3D *pLocation
);
```

Description	Sets the object's 3D location using Cartesian coordinates.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pLocation	[in]	Pointer to a vector containing the 3D location in right-handed Cartesian coordinates.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetLocationSpherical

```
SLresult (*SetLocationSpherical) (
    SL3DLocationItf self,
    SLmillidegree azimuth,
    SLmillidegree elevation,
    SLmillimeter distance
);
```

Description	Sets the object's 3D location using spherical coordinates.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	azimuth	[in]	The azimuth angle in millidegrees. The valid range is [-360000, 360000].
	elevation	[in]	The elevation angle in millidegrees. The valid range is [-90000, 90000].
	distance	[in]	The distance in millimeters from the origin. The valid range is [0, SL_MILLIMETER_MAX].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

Move

```
SLresult (*Move) (
    SL3DLocationItf self,
    const SLVec3D *pMovement
);
```

Description	Moves the object pMovement amount relative to the current location.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMovement	[in]	Pointer to a vector containing the transform in right-handed Cartesian coordinates.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	If the move would cause the object to be located outside of the SLint32 space the behavior of this method is undefined.		

GetLocationCartesian

```
SLresult (*GetLocationCartesian) (
   SL3DLocationItf self,
   SLVec3D *pLocation
);
```

Description	Gets the object's 3D location expressed in Cartesian coordinates.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pLocation`	[out]	Pointer to a vector to receive the 3D location in right-handed Cartesian coordinates. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	The accuracy of the output location is limited: one or more of the components (x, y or z) can differ by (a) up to 4% of the largest (absolute) vector component, or (b) 1 mm from the accurate value, which ever is greater.		

SetOrientationVectors

```
SLresult (*SetOrientationVectors) (
    SL3DLocationItf self,
    const SLVec3D *pFront,
    const SLVec3D *pAbove
);
```

Description	Sets the object's 3D orientation using vectors.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pFront`	[in]	Pointer to a vector specifying the *Front* vector of the object in the world coordinate system.
	`pAbove`	[in]	Pointer to a vector specifying the *Above* vector mentioned above.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	The specified vectors need not be unit vectors (for example, normalized): they can have any non-zero magnitude. Please note that there are three alternative methods for setting the orientation: `SetOrientationAngles` for setting it using angles relative to the default orientation, `SetOrientationVectors` for setting it using orientation vectors relative to the default orientation and `Rotate` for setting it using the given rotation axis and angle relative to the current orientation. If any argument is close to the zero vector; or if the specified vectors are close to parallel the `SL_RESULT_PARAMETER_INVALID` error code will be returned and the orientation of the object will remain unchanged.		

SetOrientationAngles			

`SLresult (*SetOrientationAngles) (`
`    SL3DLocationItf self,`
`    SLmillidegree heading,`
`    SLmillidegree pitch,`
`    SLmillidegree roll`
`);`			

Description	Sets the object's 3D orientation using angles.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`heading`	[in]	The angle of rotation around the y-axis in millidegrees. The valid range is [-360000, 360000].
	`pitch`	[in]	The angle of rotation around the x-axis in millidegrees. The valid range is [-90000, 90000].
	`roll`	[in]	The angle of rotation around the z-axis in millidegrees. The valid range is [-360000, 360000].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

Rotate

```
SLresult (*Rotate) (
    SL3DLocationItf self,
    SLmillidegree theta,
    const SLVec3D *pAxis
);
```

Description	Rotates the object's orientation. The rotation is theta millidegrees relative to the current orientation. This rotation follows the right-hand rule, so if the rotation axis (`pAxis`) points toward the user, the rotation will be counterclockwise.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`theta`	[in]	The amount of rotation in millidegrees.
	`pAxis`	[in]	The rotation axis. This must not be a zero vector, but the length does not matter.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

GetOrientationVectors

```
SLresult (*GetOrientationVectors) (
    SL3DLocationItf self,
    SLVec3D *pFront,
    SLVec3D *pUp
);
```

Description	Gets the object's 3D orientation as vectors.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pFront	[out]	Pointer to a vector to receive the current front orientation. This must be non-NULL. The vector will have a normalized length of approximately 1000 mm.
	pUp	[out]	Pointer to a vector to receive the current up orientation. This must be non-NULL. The vector will have a normalized length of approximately 1000 mm.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The accuracy of the output orientations is limited: one or more of the components (x, y or z) can differ by (a) up to 4% of the largest (absolute) vector component, or (b) 1 mm from the accurate value, which ever is greater.		

8.7 SL3DMacroscopicItf

Description

This interface is for controlling the size of a 3D sound source. By default, a sound source has a size of zero – that is, it is a point. This interface allows the dimensions (width, height, and depth) of a sound source to be specified so that it no longer behaves as a point source. This is useful for relatively big sound sources like waterfalls. The orientation of the macroscopic volume also can be specified. See the Orientation section in 8.5 for detailed explanation of different methods for setting the orientation.

The following diagram illustrates how the dimensions map to the orientation vectors of the 3D source.

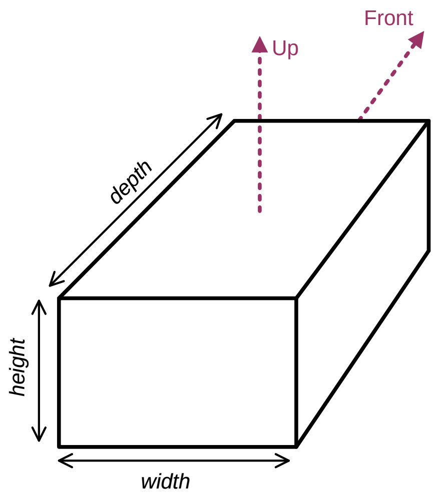


Figure 29: Dimensions-orientation vectors

It is good practice to use the macroscopic effect only on selected, relatively big 3D sources, on which the effect will be most effective or noticeable, since macroscopic effect processing can use additional memory and processing power. Developers can conserve resources by exposing this interface only on 3D sources that require the macroscopic effect.

Please note that you can use the `SL3DLocationItf` interface to locate the 3D sound source; the location defined by `3DLocationItf` is the location of the center of the macroscopic sound source.

Please note that you can use the `SL3DSourceItf` interface to specify the distance attenuation model of the 3D sound source. The exact distance used for distance attenuation calculation in the case of a macroscopic source is implementation-dependent and the implementation can take into account the macroscopicity near the sound source. The application should specify the distance attenuation as it would in the case where the source is a point source. Relatively far from the macroscopic sound source, the distance attenuation behaves similarly to non-macroscopic sources.

Please note that you can use the SL3DSourceItf interface to specify the directivity model of the 3D sound source. Relatively far from the macroscopic sound source, the directivity-based attenuation behaves similarly to non-macroscopic sources. The exact directivity-based attenuation near the macroscopic sound source is implementation-dependent and the implementation can take into account the macroscopicity when the sound source is near the listener. The application should specify the directivity as it would in cases where the source is a point source.

This interface can be exposed on the 3DGroup object, if macroscopicity is supported.

The following restriction must be adhered to when exposing this interface on a player:

- This interface can be exposed on a player only if the SL3DLocationItf interface is exposed on the same player. This interface can be exposed on a player at creation as long as the SL3DLocationItf is also exposed at creation.

Parameter changes made using this interface may be deferred using the SL3DCommitItf interface [see section 8.1].

Prototype

```
SL_API extern const SLInterfaceID SL_IID_3DMACROSCOPIC;

struct SL3DMacroscopicItf_;
typedef const struct SL3DMacroscopicItf_ * const * SL3DMacroscopicItf;

struct SL3DMacroscopicItf_ {
   SLresult (*SetSize) (
         SL3DMacroscopicItf self,
         SLmillimeter width,
         SLmillimeter height,
         SLmillimeter depth
   );
   SLresult (*GetSize) (
         SL3DMacroscopicItf self,
         SLmillimeter *pWidth,
         SLmillimeter *pHeight,
         SLmillimeter *pDepth
   );
   SLresult (*SetOrientationAngles) (
         SL3DMacroscopicItf self,
         SLmillidegree heading,
         SLmillidegree pitch,
         SLmillidegree roll
   );
```

```
      SLresult (*SetOrientationVectors) (
            SL3DMacroscopicItf self,
            const SLVec3D *pFront,
            const SLVec3D *pAbove
      );
      SLresult (*Rotate) (
            SL3DMacroscopicItf self,
            SLmillidegree theta,
            const SLVec3D *pAxis
      );
      SLresult (*GetOrientationVectors) (
            SL3DMacroscopicItf self,
            SLVec3D *pFront,
            SLVec3D *pUp
      );
   };
```

Interface ID

5089aec0-ddd3-11db-9ad3-0002a5d5c51b

Defaults

Size (width, height, depth): (0 mm, 0 mm, 0 mm) – a point

Front orientation (x, y, z): (0, 0, -1000) – looking forward

Up orientation (x, y, z): (0, 1000, 0) - looking up

That is, the initial position is the origin and the initial orientation is towards the negative Z-axis, up-direction being towards the positive Y-axis. In rotation angles, this equals: heading = 0 degrees; pitch = 0 degrees and roll = 0 degrees.

Methods

SetSize			
```
SLresult (*SetSize)(
   SL3DMacroscopicItf self,
   SLmillimeter width,
   SLmillimeter height,
   SLmillimeter depth
);
```
Description	Sets the size of the 3D sound source.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	`width`	[in]	The “width” of the sound source in its transformed X (or “right”) dimension, in millimeters. The valid range in [0, `SL_MILLIMETER_MAX`].
	`height`	[in]	The “height” of the sound source in its transformed Y (or “up”) dimension, in millimeters. The valid range in [0, `SL_MILLIMETER_MAX`].
	`depth`	[in]	The “thickness” or “depth” of the sound source in its transformed Z (or “front”) dimension, in millimeters. The valid range in [0, `SL_MILLIMETER_MAX`].

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

GetSize

```
SLresult (*GetSize)(
   SL3DMacroscopicItf self,
   SLmillimeter *pWidth,
   SLmillimeter *pHeight,
   SLmillimeter *pDepth
);
```

Description	Gets the size of the 3D sound source.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pWidth	[out]	The “width” of the sound source in its transformed X (or “right”) dimension, in millimeters. This parameter must be non-NULL.
	pHeight	[out]	The “height” of the sound source in its transformed Y (or up) dimension, in millimeters. This parameter must be non-NULL.
	pDepth	[out]	The “thickness” or “depth” of the sound source in its transformed Z (or front) dimension, in millimeters. This parameter must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetOrientationAngles

```
SLresult (*SetOrientationAngles)(
    SL3DMacroscopicItf self,
    SLmillidegree heading,
    SLmillidegree pitch,
    SLmillidegree roll
);
```

Description	Sets the 3D orientation of the macroscopic volume using angles.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`heading`	[in]	The rotation around the Y-axis of the object, in millidegrees. The valid range is [-360000, 360000].
	`pitch`	[in]	The rotation around the X-axis of the object, in millidegrees. The valid range is [-90000, 90000].
	`roll`	[in]	The rotation around the Z-axis of the object, in millidegrees. The valid range is [-360000, 360000].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Please note that there are three alternative methods for setting the orientation: `setOrientationAngles` for setting it using angles relative to the default orientation, `setOrientationVectors` for setting it using orientation vectors relative to the default orientation, and `Rotate` for setting it using the given rotation axis and angle relative to the current orientation.		

SetOrientationVectors			
`SLresult (*SetOrientationVectors)( SL3DMacroscopicItf self, const SLVec3D *pFront, const SLVec3D *pAbove );`			
Description	Sets the 3D orientation of the macroscopic volume using vectors. The specified vectors need not be unit vectors (that is, normalized): they can have any non-zero magnitude.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pFront`	[in]	Pointer to a vector specifying the front vector of the object in the world coordinate system.
	`pAbove`	[in]	Pointer to a vector specifying the "above" vector mentioned above.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Please note that there are three alternative methods for setting the orientation: `SetOrientationAngles` for setting it using angles relative to the default orientation, `SetOrientationVectors` for setting it using orientation vectors relative to the default orientation and `Rotate` for setting it using the given rotation axis and angle relative to the current orientation. If any argument is close to the zero vector; or if the specified vectors are close to parallel the `SL_RESULT_PARAMETER_INVALID` error code will be returned and the orientation of the object will remain unchanged.		

Rotate

```
SLresult (*Rotate) (
    SL3DMacroscopicItf self,
    SLmillidegree theta,
    const SLVec3D *pAxis
);
```

Description	Rotates the macroscopic volume's orientation. The rotation is theta millidegrees relative to the current orientation. This rotation follows the right-hand rule, so if the rotation axis (`pAxis`) points toward the user, the rotation will be counterclockwise.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`theta`	[in]	The amount of rotation in millidegrees.
	`pAxis`	[in]	The rotation axis. This must not be a zero vector, but the length does not matter.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	Please note that there are three alternative methods for setting the orientation: `setOrientationAngles` for setting it using angles relative to the default orientation, `setOrientationVectors` for setting it using orientation vectors relative to the default orientation and `Rotate` for setting it using the given rotation axis and angle relative to the current orientation.		

GetOrientationVectors

```
SLresult (*GetOrientationVectors)(
    struct SL3DMacroscopicItf self,
    SLVec3D *pFront,
    SLVec3D *pUp
);
```

Description	Gets the 3D orientation of the macroscopic volume.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pFront	[out]	Pointer to a vector to receive the current front orientation. The vector will have a normalized length of approximately 1000 mm. This must be non-NULL.
	pUp	[out]	Pointer to a vector to receive the current up orientation. The vector will have a normalized length of approximately 1000 mm. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The accuracy of the output orientations is limited: one or more of the components (x, y or z) can differ by (a) up to 4% of the largest (absolute) vector component, or (b) 1 mm from the accurate value, which ever is greater.		

8.8 SL3DSourceItf

Description

This interface controls 3D parameters that are unique to 3D sources. If this interface is exposed on a 3D group, the methods control the 3D characteristics of all the players in the 3D group.

The following restriction must be adhered to when exposing this interface on a player object:

- This interface can be exposed on a player only if the `SL3DLocationItf` interface is exposed on the same player. This interface can be exposed on a player at creation as long as the `SL3DLocationItf` is also exposed at creation.

Parameter changes made using this interface may be deferred using the `SL3DCommitItf` interface [see section 8.1].

This interface is supported on the Audio Player [see section 7.2], MIDI Player [see section 7.8] and 3D Group [see section 7.8] objects.

See section B.5.2 for an example using this interface.

Head Relative

In most cases, the listeners and 3D sources move independently of each other. However, in some cases a sound source may track the listener’s position. Consider, for example, a first person game where the listener and camera view are identical. As the listener moves, any footsteps the listener makes are likely to track the listener at a fixed position.

OpenSL ES provides support for this behavior by allowing the developer to register a 3D source as *head relative*. When a 3D source is in head relative mode, its location, velocity and orientation are specified relative to the listener’s location, velocity and orientation. The consequence of this is that as the listener moves, the 3D source will stay at a constant distance from the listener when the 3D source’s location is not changed.

Distance Rolloff

The level of sound source heard by a listener decreases as a sound source moves further away. This is called *distance rolloff*. OpenSL ES provides several methods to control a 3D source’s rolloff characteristics.

The starting point for the OpenSL ES distance rolloff model is a *minimum distance* that must be specified for each 3D source (or left at the default of one meter). This is the distance at which the gain due to rolloff is constant at unity (that is, the 3D source is neither attenuated nor amplified). The minimum distance is used for representing the *size* (in audio-centric terms) of the sound source the 3D source represents. The minimum distance is required because it is standard practice of sound designers to normalize the audio data. Without any adjustment of overall 3D source's minimum distance, a sound sample of an insect would be percieved as loud as that of a train engine at a given distance.

A developer can also control the *maximum distance* of a 3D source. This is the distance at which the 3D source's distance rolloff gain is clamped or muted altogether, depending on the *mute at maximum distance* setting.

The minimum and maximum distances control the distances at which attenuation is applied, but they do not control the rate of attenuation. For this the developer can set the distance rolloff model and its rolloff factor.

OpenSL ES supports two different distance rolloff models: an exponential rolloff model, in which the level of sound decays at an exponential rate due to distance and a linear rolloff model, in which the sound decays at a linear rate due to distance.

The exponential distance rolloff model is defined as follows:

$$G_d = \begin{cases} 1 & \text{if } d < d_{\min} \\ 0 & \text{if } d \geq d_{\max} \text{and } \mathit{rolloffMaxDistanceMute} = \mathit{true} \\ \left(\dfrac{d_{\min}}{d_{\max}}\right)^{\mathit{rolloffFactor}} & \text{if } d \geq d_{\max} \text{and } \mathit{rolloffMaxDistanceMute} = \mathit{false} \\ \left(\dfrac{d_{\min}}{d}\right)^{\mathit{rolloffFactor}} & \text{otherwise} \end{cases}$$

where:

- G_d is the linear gain due to distance rolloff at distance d.
- d is the distance between the 3D source and the listener. This is calculated as the length of the vector that is formed by subtracting the listener's location vector from the 3D source's location vector (or when in head relative mode, simply the length of the 3D source's location vector). If the `SL3DLocationItf` interface is not exposed on the listener, the default listener location is used in this calculation.
- $d_{\min}$ is the minimum distance in millimeters; the distance from the listener within which the 3D source gain is constant. This distance is set in `SetRolloffDistances()`.
- $d_{\max}$ is the maximum distance in millimeters; the distance from the listener at which the 3D source gain is no longer attenuated due to distance. This distance is set in `SetRolloffDistances()`.

- *rolloffFactor* is the rate at which the gain attenuates due to distance. This is set using a permille scale in `SetRolloffFactor()`. 1000 ‰ is equal to a rolloff factor of one.
- *rolloffMaxDistanceMute* controls whether the 3D source is muted when its distance from the listener is beyond the maximum distance. This is set using `SetRolloffMaxDistanceMute()`.

The exponential distance model closely matches the distance effect in the real world. The *rolloff factor* controls the rate of decay. The graph below shows the effect of the rolloff factor at 0.5 (500 ‰), one (1000 ‰), the default rolloff factor, and two (2000 ‰). In this example, the minimum distance is 1000 mm, the maximum distance is 12000 mm and the 3D source is configured to mute at the maximum distance.

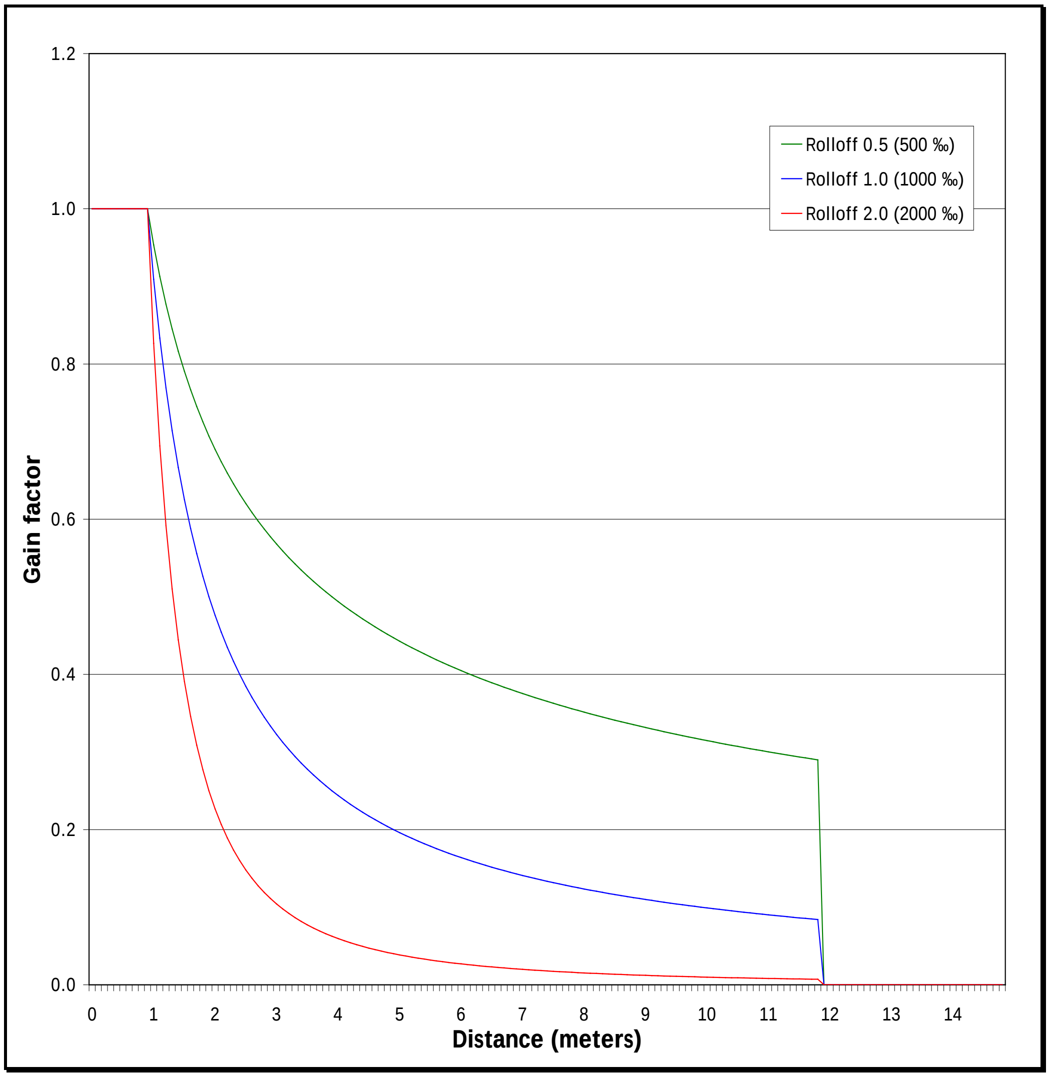


Figure 30: Rolloff factor effect

OpenSL ES also supports a linear distance model. This does not accurately model real-world distance rolloff, but can be useful for some games. The linear distance rolloff model is defined as follows:

$$G_d = \begin{cases} 1 & \text{if } d < d_{\min} \\ 0 & \text{if } d \geq d_{\max} \text{ and } \\ & \quad \mathit{rolloffMaxDistanceMute} = \mathit{true} \\ \max(0, 1 - \mathit{rolloffFactor}) & \text{if } d \geq d_{\max} \text{and} \\ & \quad \mathit{rolloffMaxDistanceMute} = \mathit{false} \\ \max\left(0, 1 - \left(\mathit{rolloffFactor} \times \dfrac{d - d_{\min}}{d_{\max} - d_{\min}}\right)\right) & \text{otherwise} \end{cases}$$

where:

- G_d is the linear gain due to distance rolloff at distance d.
- d is the distance between the 3D source and the listener. This is calculated as the length of the vector that is formed by subtracting the listener's location vector from the 3D source's location vector (or when in head relative mode, simply the length of the 3D source's location vector). If the `SL3DLocationItf` interface is not exposed on the listener, the default listener location is used in this calculation.
- $d_{\min}$ is the minimum distance in millimeters; the distance from the listener within which the 3D source gain is constant. This distance is set in `SetRolloffDistances()`.
- $d_{\max}$ is the maximum distance in millimeters; the distance from the listener at which the 3D source gain is no longer attenuated due to distance. This distance is set in `SetRolloffDistances()`.
- $\mathit{rolloffFactor}$ is the rate at which the gain attenuates due to distance. This is set using a permille scale in `SetRolloffFactor()`. `1000 ‰` is equal to a rolloff factor of one.
- $\mathit{rolloffMaxDistanceMute}$ controls whether the 3D source is muted when its distance from the listener is beyond the maximum distance. This is set using `SetRolloffMaxDistanceMute()`.

The graph below shows the effect of different rolloff factors when using the linear rolloff model. In this example, the minimum distance is 1000 mm, the maximum distance is 12000 mm and the 3D source is configured to mute at the maximum distance.

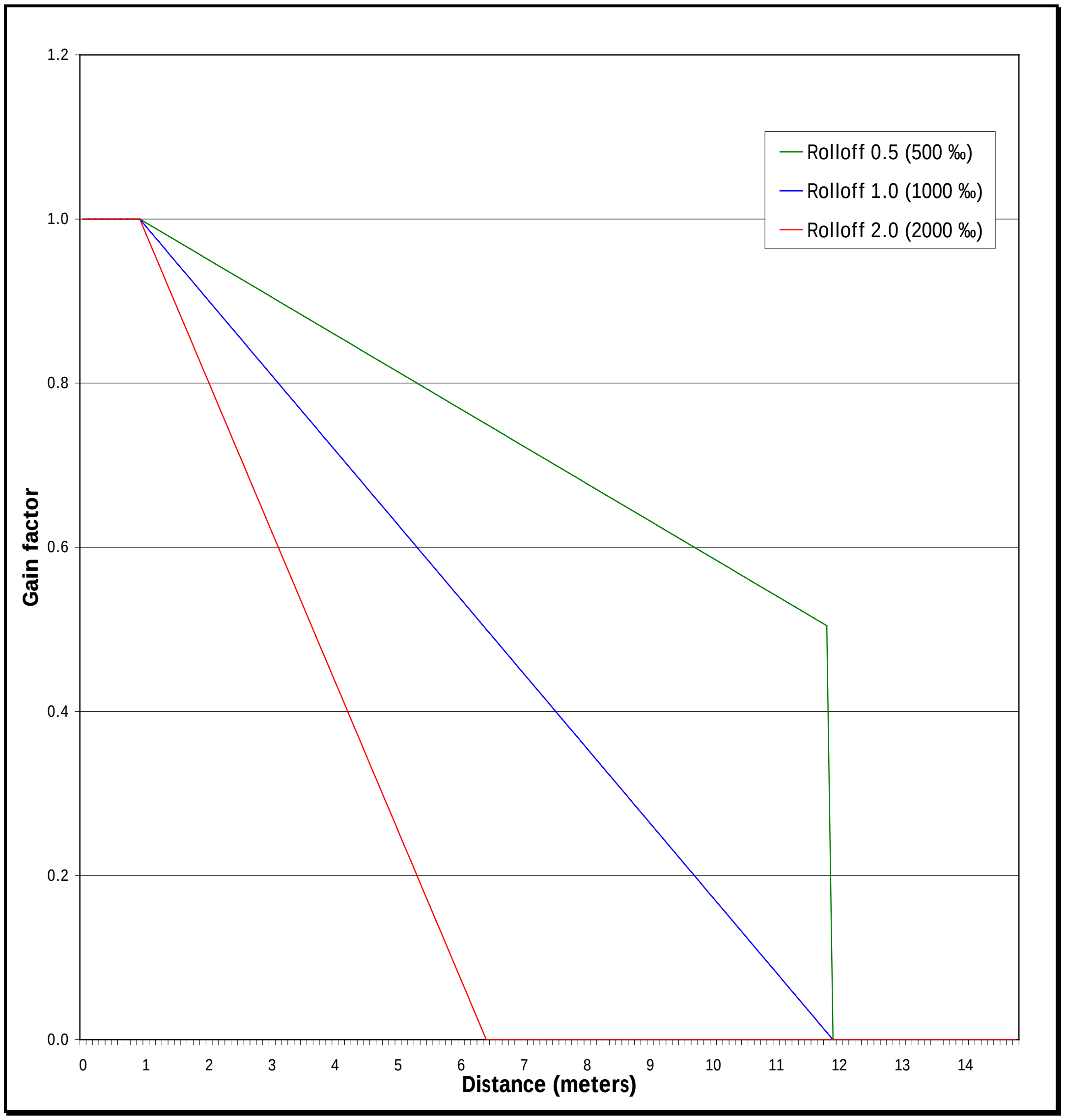


Figure 31: Linear rolloff

Room Rolloff

The equations above show how to calculate a 3D source's direct path distance rolloff. These same equations are also used for calculating a 3D source's reverb path distance rolloff, but using the *room rolloff factor* (set using `SetRoomRolloffFactor()`) instead of the standard rolloff factor. This allows an application to control the rolloff rate for a sound source's contribution to a reverb environment separately from its direct path contribution.

The minimum and maximum rolloff distances are the same for both the direct path and reverb path.

Sound Cones

Many sound sources are omnidirectional, that is, they radiate sound equally in all directions so that they sound exactly the same no matter what their orientation. An example would be an exploding bomb. Other sound sources are more accurately represented by a 3D source that radiates more sound in one direction than in others(as, for example, a human voice), which projects more forwards than in other directions.

Real life radiation patterns for directional sources are complex, but a good effect can be created using the concept of the "sound cone", the axis of which defines the direction of strongest radiation, as shown in the diagram below.

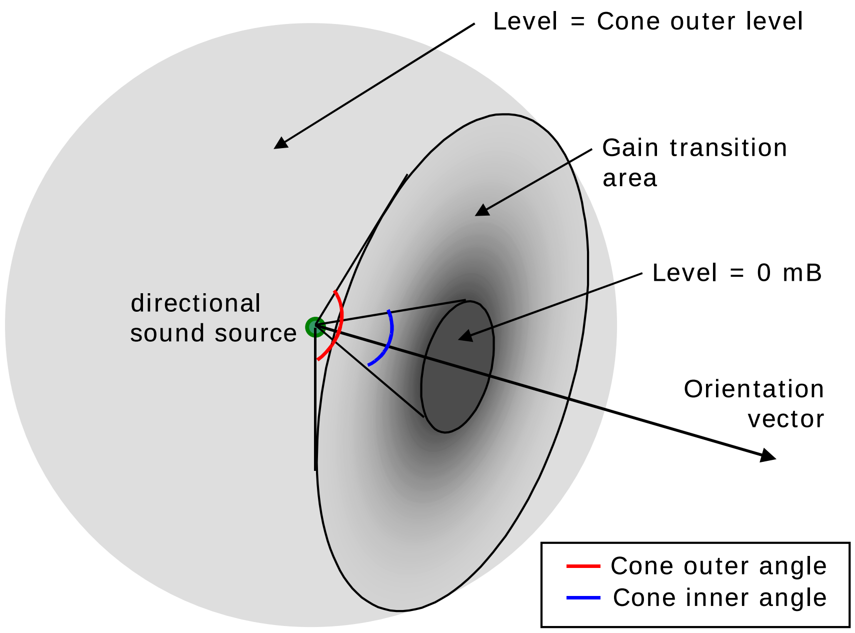


Figure 32: Sound cone

The *cone inner* and *cone outer angles* define the inner and outer sound cones, respectively. Within the inner cone, the sound source's level is constant, with no attenuation applied. Outside the outer cone, the sound source's level has an attenuation equal to the *cone outer level*. Between the inner and outer cones, perceived sound decreases linearly as the angle of the listener from the axis of orientation of the sound source increases. If `SL3DLocationItf` is not exposed on the listener, the default listener orientation is assumed for the listener.

If the cone outer angle is not much larger than the cone inner angle, there will be a relatively sudden change in volume as the listener moves through these angles. This would be appropriate for some kind of weapon that emits a narrow beam, for example. A useful optical analogy is a lighthouse, which emits a beam of light. The wider the beam, the larger the cone inner angle should be. The sharper the focus of the edge of the beam, the closer the cone outer angle should be to the cone inner angle.

One way to approach about cone outer level is to think of the directional and omnidirectional components of the sound source. If a cone outer level close to 0 mB is specified, most of the 3D source’s power will be omnidirectional. Conversely, if a value close to silence is specified, most of the power will be directional.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_3DSOURCE;

struct SL3DSourceItf_;
typedef const struct SL3DSourceItf_ * const * SL3DSourceItf;

struct SL3DSourceItf_ {
   SLresult (*SetHeadRelative) (
         SL3DSourceItf self,
         SLboolean headRelative
   );
   SLresult (*GetHeadRelative) (
         SL3DSourceItf self,
         SLboolean *pHeadRelative
   );
   SLresult (*SetRolloffDistances) (
         SL3DSourceItf self,
         SLmillimeter minDistance,
         SLmillimeter maxDistance
   );
   SLresult (*GetRolloffDistances) (
         SL3DSourceItf self,
         SLmillimeter *pMinDistance,
         SLmillimeter *pMaxDistance
   );
   SLresult (*SetRolloffMaxDistanceMute) (
         SL3DSourceItf self,
         SLboolean mute
   );
   SLresult (*GetRolloffMaxDistanceMute) (
         SL3DSourceItf self,
         SLboolean *pMute
   );
   SLresult (*SetRolloffFactor) (
         SL3DSourceItf self,
         SLpermille rolloffFactor
   );
   SLresult (*GetRolloffFactor) (
         SL3DSourceItf self,
         SLpermille *pRolloffFactor
   );
```

```
      SLresult (*SetRoomRolloffFactor) (
            SL3DSourceItf self,
            SLpermille roomRolloffFactor
      );
      SLresult (*GetRoomRolloffFactor) (
            SL3DSourceItf self,
            SLpermille *pRoomRolloffFactor
      );
      SLresult (*SetRolloffModel) (
            SL3DSourceItf self,
            SLuint8 model
      );
      SLresult (*GetRolloffModel) (
            SL3DSourceItf self,
            SLuint8 *pModel
      );
     SLresult (*SetCone) (
            SL3DSourceItf self,
            SLmillidegree innerAngle,
            SLmillidegree outerAngle,
            SLmillibel outerLevel
      );
      SLresult (*GetCone) (
            SL3DSourceItf self,
            SLmillidegree *pInnerAngle,
            SLmillidegree *pOuterAngle,
            SLmillibel *pOuterLevel
      );
   };
```

Interface ID

70bc7b00-ddd3-11db-a873-0002a5d5c51b

Defaults

Head Relative: SL_BOOLEAN_FALSE

Max Distance Mute: SL_BOOLEAN_FALSE

Max Distance: SL_MILLIMETER_MAX

Min Distance: 1000 mm

Cone Angles (inner, outer): (360000 mdeg, 360000 mdeg)

Cone Outer Level: 0 mB

Rolloff Factor: 1000 ‰

Room rolloff factor: 0 ‰

Distance Model: SL_ROLLOFFMODEL_EXPONENTIAL

Methods

SetHeadRelative			
`SLresult (*SetHeadRelative) ( SL3DSourceItf self, SLboolean headRelative );`			
Description	Sets whether the 3D source should be treated as head relative.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	headRelative	[in]	If true, the 3D source is considered head relative: the properties of the 3D source's location, velocity and orientation are treated as relative to the listener rather than the origin. If false, these properties of the 3D source are treated as relative to the origin.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS		
Comments	When headRelative is equal to true, sound sources track the listener's position (such as footsteps).		

GetHeadRelative			
`SLresult (*GetHeadRelative) ( struct SL3DSourceItf self, SLboolean *pHeadRelative );`			
Description	Gets the 3D source's head relative state.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pHeadRelative	[out]	Pointer to a location to receive a Boolean signifying whether or not the 3D source is in head relative mode. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetRolloffDistances

```
SLresult (*SetRolloffDistances) (
    SL3DSourceItf self,
    SLmillimeter minDistance,
    SLmillimeter maxDistance
);
```

Description	Sets the minimum and maximum distances of the 3D source. The minimum distance is the distance from the listener within which the 3D source gain is constant. The maximum distance is the distance from the listener at which the 3D source gain is no longer attenuated due to distance.		
Pre-conditions	minDistance must be <= maxDistance.		
Parameters	`self`	[in]	Interface self-reference.
	`minDistance`	[in]	The minimum distance of the 3D source. The valid range is (0, `SL_MILLIMETER_MAX`].
	`maxDistance`	[in]	The distance in millimeters at which the 3D source's gain is no longer attenuated due to distance. The value `SL_MILLIMETER_MAX` is used where the 3D source continues to attenuate, whatever the distance. The valid range is [`minDistance`, `SL_MILLIMETER_MAX`].

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

GetRolloffDistances			
`SLresult (*GetRolloffDistances) ( struct SL3DSourceItf self, SLmillimeter *pMinDistance, SLmillimeter *pMaxDistance );`			
Description	Gets the 3D source's minimum and maximum distances.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pMinDistance`	[out]	Pointer to a location to receive the minimum distance for the 3D source. This must be non-`NULL`.
	`pMaxDistance`	[out]	Pointer to a location to receive the maximum distance for the 3D source. This must be non-`NULL`.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetRolloffMaxDistanceMute			
`SLresult (*SetRolloffMaxDistanceMute) ( SL3DSourceItf self, SLboolean mute );`			
Description	Sets whether the 3D source is muted when beyond the maximum rolloff distance.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`mute`	[in]	If true, the 3D source is muted when it is more than the maximum distance away from the listener; otherwise, it is not muted.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS		
Comments	None.		

GetRolloffMaxDistanceMute

```
SLresult (*GetRolloffMaxDistanceMute) (
    struct SL3DSourceItf self,
    SLboolean *pMute
);
```

Description	Gets whether 3D source is muted when beyond the maximum rolloff distance.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMute	[out]	Pointer to a location to receive a Boolean signifying whether the 3D source is muted beyond the maximum distance. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetRolloffFactor

```
SLresult (*SetRolloffFactor) (
    SL3DSourceItf self,
    SLpermille rolloffFactor
);
```

Description	Sets the distance rolloff factor for the 3D source.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	rolloffFactor	[in]	The rolloff factor in permille. The valid range is [0, 10000].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

GetRolloffFactor

```
SLresult (*GetRolloffFactor) (
   struct SL3DSourceItf self,
   SLpermille *pRolloffFactor
);
```

Description	Gets the distance rolloff factor for the 3D source.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pRolloffFactor`	[out]	Pointer to a location to receive the rolloff factor. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

SetRoomRolloffFactor

```
SLresult (*SetRoomRolloffFactor) (
   SL3DSourceItf self,
   SLpermille roomRolloffFactor
);
```

Description	Sets the room rolloff factor for the 3D source.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`roomRolloffFactor`	[in]	The room rolloff factor in permille. The valid range is [0, 10000].

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

GetRoomRolloffFactor

```
SLresult (*GetRoomRolloffFactor) (
    struct SL3DSourceItf self,
    SLpermille *pRoomRolloffFactor
);
```

Description	Gets the distance room rolloff factor for the 3D source.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pRoomRolloffFactor	[out]	Pointer to a location to receive the room rolloff factor. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetRolloffModel

```
SLresult (*SetRolloffModel) (
    SL3DSourceItf self,
    SLuint8 model
);
```

Description	Sets the distance rolloff model used for calculating decay due to distance for the 3D source.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	model	[in]	The distance models. The standard distance models supported by all implementations are: SL_ROLLOFFMODEL_LINEAR, SL_ROLLOFFMODEL_EXPONENTIAL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

GetRolloffModel

```
SLresult (*GetRolloffModel) (
    struct SL3DSourceItf self,
    SLuint8 *pModel
);
```

Description	Gets the distance rolloff model for the 3D source.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pModel	[out]	Pointer to location to receive the distance model. The standard distance models supported by all implementations are: SL_ROLLOFFMODEL_LINEAR, SL_ROLLOFFMODEL_EXPONENTIAL. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetCone

```
SLresult (*SetCone) (
    SL3DSourceItf self,
    SLmillidegree innerAngle,
    SLmillidegree outerAngle,
    SLmillibel outerLevel
);
```

Description	Sets the sound cones for the 3D source.		
Pre-conditions	innerAngle must be <= outerAngle.		
Parameters	self	[in]	Interface self-reference.
	innerAngle	[in]	Inner cone angle in millidegrees. Its value should be in the range [0, 360000].
	outerAngle	[in]	Outer cone angle in millidegrees. Its value should be in the range [0, 360000].
	outerLevel	[in]	Outer cone volume in millibels. Its value should be in the range [SL_MILLIBEL_MIN, 0]. The special value SL_MILLBEL_MIN indicates that the cone outer volume should be silent.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PRECONDITIONS_VIOLATED SL_RESULT_PARAMETER_INVALID		
Comments	None.		

GetCone

```
SLresult (*GetCone) (
   struct SL3DSourceItf self,
   SLmillidegree *pInnerAngle,
   SLmillidegree *pOuterAngle,
   SLmillibel *pOuterLevel
);
```

Description	Gets the sound cones for the 3D source.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pInnerAngle`	[out]	Pointer to location to receive the inner cone angle. This must be non-`NULL`.
	`pOuterAngle`	[out]	Pointer to location to receive the outer cone angle. This must be non-`NULL`.
	`pOuterLevel`	[out]	Pointer to a location to receive the cone outer level in millibels. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

8.9 SLAudioDecoderCapabilitiesItf

Description

This interface provides methods for querying the audio decoding capabilities of the audio engine.

This interface provides a means of enumerating all audio decoders available on an engine where a decoderId represents each decoder. It also provides a means to query the capabilities of each decoder. A given decoder may support several profile/mode pairs each with their own capabilities (such as maximum sample rate or bit rate) appropriate to that profile and mode pair. Therefore, this interface represents the capabilities of a particular decoder as a list of capability entries queriable by decoderID and capability entry index.

The set of audio decoders supported by the engine does not change during the lifetime of the engine though dynamic resource constraints may limit actual availability when an audio decoder is requested.

This interface is supported on engine objects [see section 7.4].

Prototype

```
SL_API extern const SLInterfaceID SL_IID_AUDIODECODERCAPABILITIES;

struct SLAudioDecoderCapabilitiesItf_;
typedef const struct SLAudioDecoderCapabilitiesItf_
    * const * SLAudioDecoderCapabilitiesItf;

struct SLAudioDecoderCapabilitiesItf_ {
   SLresult (*GetAudioDecoders) (
         SLAudioDecoderCapabilitiesItf self,
         SLuint32 * pNumDecoders ,
         SLuint32 *pDecoderIds
   );
   SLresult (*GetAudioDecoderCapabilities) (
         SLAudioDecoderCapabilitiesItf self,
         SLuint32 decoderId,
         SLuint32 *pIndex,
         SLAudioCodecDescriptor *pDescriptor
   );
};
```

Interface ID

3fe5a3a0-fcc6-11db-94ac-0002a5d5c51b

Defaults

Not applicable.

Methods

GetAudioDecoders			
`SLresult (*GetAudioDecoders) ( SLAudioDecoderCapabilitiesItf self, SLuint32 *pNumDecoders, SLuint32 *pDecoderIds );`			
Description	Retrieves the available audio decoders.		
Pre-conditions	None.		
Post-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pNumDecoders	[in/out]	If pDecoderIds is NULL, pNumDecoders returns the number of decoders available. Returns 0 if there are no decoders. If pDecodersIds is non-NULL, as an input pNumDecoders specifies the size of the pDecoderIds array and as an output it specifies the number of decoder IDs available within the pDecoderIds array.
	pDecoderIds	[out]	Array of audio decoders provided by the engine. Refer to SL_AUDIOCODEC defines [see section 9.2.1].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	GetAudioDecoderCapabilities()		

GetAudioDecoderCapabilities			
`SLresult (*GetAudioDecoderCapabilities) ( SLAudioDecoderCapabilitiesItf self, SLuint32 decoderId, SLuint32 *pIndex, SLAudioCodecDescriptor *pDescriptor );`			
Description	Queries for the audio decoder capabilities.		
Pre-conditions	None.		
Post-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	decoderId	[in]	Identifies the supported audio decoder. Refer to SL_AUDIOCODEC defines [see section 9.2.1].
	pIndex	[in/out]	If pDescriptor is NULL, pIndex returns the number of capabilities structures (one per profile/mode pair of the decoder). Returns 0 if there are no capabilities. If pDescriptor is non-NULL, pIndex is an index used for enumerating capabilities structures. Supported index range is 0 to N-1, where N is the number of capabilities structures, one for each profile/mode pair of the decoder.
	pDescriptor	[out]	Pointer to structure defining the capabilities of the audio decoder. There is one structure per profile/mode pair of the decoder.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This method outputs a structure that contains one or more pointers to arrays. If any pointers are NULL, the corresponding size field of the array will be the size needed to allocate the array.		
See also	GetAudioDecoders()		

8.10 SLAudioEncoderItf

Description

This interface is used for setting the parameters to be used by an audio encoder. It is realized on a media object with audio encoding capabilities, such as an audio recorder. Once the supported codecs have been enumerated using `SLAudioEncoderCapabilitiesItf` on the engine [see section 8.11], the encoding settings can be set using this interface.

This interface is supported on Audio Recorder objects [see section 7.3].

Prototype

```
SL_API extern const SLInterfaceID SL_IID_AUDIOENCODER;

struct SLAudioEncoderItf_;
typedef const struct SLAudioEncoderItf_ * const * SLAudioEncoderItf;

struct SLAudioEncoderItf_ {
   SLresult (*SetEncoderSettings) (
         SLAudioEncoderItf       self,
         SLAudioEncoderSettings  *pSettings
   );
   SLresult (*GetEncoderSettings) (
         SLAudioEncoderItf       self,
         SLAudioEncoderSettings  *pSettings
   );
};
```

Interface ID

d0897d20-f774-11db-b80d-0002a5d5c51b

Defaults

No default settings are mandated.

Methods

SetEncoderSettings

```
SLresult (*SetEncoderSettings) (
  SLAudioEncoderItf        self,
  SLAudioEncoderSettings   *pSettings
);
```

Description	Set audio encoder settings.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pSettings	[in]	Specifies the audio encoder settings to be applied [see section 9.1.2].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	GetEncoderSettings()		

GetEncoderSettings

```
SLresult (*GetEncoderSettings) (
  SLAudioEncoderItf        self,
  SLAudioEncoderSettings      *pSettings
);
```

Description	Get audio encoder settings.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pSettings	[out]	Specifies a pointer to the structure that will return the audio encoder settings [see section 9.1.2].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	SetEncoderSettings()		

8.11 SLAudioEncoderCapabilitiesItf

Description

This interface provides methods for querying the audio encoding capabilities of the audio engine.

This interface provides a means of enumerating all audio encoders available on an engine where an encoderId represents each encoder. It also provides a means to query the capabilities of each encoder. A given encoder may support several profile/mode pairs, each with their own capabilities (such as maximum sample rate or bit rate) appropriate to that profile and mode pair. Therefore, this interface represents the capabilities of a particular encoder as a list of capability entries queriable by encoderID and capability entry index.

The set of audio encoders supported by the engine does not change during the lifetime of the engine though dynamic resource constraints may limit actual availability when an audio encoder is requested.

This interface is also mandated on the audio recorder objects so that the effective capabilities of the encoder can be determined when it is part of the recorder object and source/sink combination. For example, an encoder might have an input sampling rate range of 8 kHz - 48 kHz overall, but when a recorder object is connected to a low-cost microphone, the effective sampling rate range of the encoder might only be 8 kHz - 16 kHz.

This interface is supported on engine objects [see section 7.4] and the audio recorder objects [see section 7.3].

Prototype

```
SL_API extern const SLInterfaceID SL_IID_AUDIOENCODERCAPABILITIES;

struct SLAudioEncoderCapabilitiesItf_;
typedef const struct SLAudioEncoderCapabilitiesItf_
    * const * SLAudioEncoderCapabilitiesItf;
```

```
struct SLAudioEncoderCapabilitiesItf_ {
   SLresult (*GetAudioEncoders) (
         SLAudioEncoderCapabilitiesItf self,
         SLuint32 *pNumEncoders,
         SLuint32 *pEncoderIds
   );
   SLresult (*GetAudioEncoderCapabilities) (
         SLAudioEncoderCapabilitiesItf self,
         SLuint32 encoderId,
         SLuint32 *pIndex,
         SLAudioCodecDescriptor *pDescriptor
   );
};
```

Interface ID

0f52a340-fcd1-11db-a993-0002a5d5c51b

Defaults

Not applicable.

Methods

GetAudioEncoders

```
SLresult (*GetAudioEncoders) (
  SLAudioEncoderCapabilitiesItf self,
  SLuint32 * pNumEncoders,
  SLuint32 *pEncoderIds
);
```

Description	Queries the supported audio encoders.		
Pre-conditions	None.		
Post-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pNumEncoders	[in/out]	If pEncoderIds is NULL, pNumEncoders returns the number of encoders available. Returns 0 if there are no encoders. If pEncodersIds is non-NULL, as an input pNumEncoders specifies the size of the pEncoderIds array and as an output it specifies the number of encoder IDs available within the pEncoderIds array.
	pEncoderIds	[out]	Array of audio encoders provided by the engine. Refer to SL_AUDIOCODEC defines [see section 9.2.1].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_BUFFER_INSUFFICIENT		
Comments	None.		
See also	GetAudioEncoderCapabilities ()		

GetAudioEncoderCapabilities			
`SLresult (*GetAudioEncoderCapabilities) ( SLAudioEncoderCapabilitiesItf self, SLuint32 encoderId, SLuint32 *pIndex, SLAudioCodecDescriptor *pDescriptor );`			
Description	Queries for the audio encoder's capabilities.		
Pre-conditions	None.		
Post-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	encoderId	[in]	Identifies the supported audio encoder. Refer to SL_AUDIOCODEC defines [see section 9.2.1].
	pIndex	[in/out]	If pCapabilities is NULL, pIndex returns the number of capabilities. Returns 0 if there are no capabilities. If pCapabilities is non-NULL, pIndex is an index used for enumerating profiles. Supported index range is 0 to N-1, where N is the number of capabilities of the encoder.
	pDescriptor	[out]	Pointer to structure containing the capabilities of the audio encoder [see section 9.1.1].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This method outputs a structure that contains one or more pointers to arrays. If any pointers are NULL, the corresponding size field of the array will be the size needed to allocate the array.		
See also	GetAudioDecoders()		

8.12 SLAudioIODeviceCapabilitiesItf

Description

This interface is for enumerating the audio I/O devices on the platform and for querying the capabilities and characteristics of each available audio I/O device.

This interface is supported on the engine object [see section 7.4].

See Appendix C: for examples using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_AUDIOIODEVICECAPABILITIES;

struct SLAudioIODeviceCapabilitiesItf_;
typedef const struct SLAudioIODeviceCapabilitiesItf_
         * const * SLAudioIODeviceCapabilitiesItf;

struct SLAudioIODeviceCapabilitiesItf_ {
   SLresult (*GetAvailableAudioInputs)(
         SLAudioIODeviceCapabilitiesItf self,
         SLuint32 *pNumInputs,
         SLuint32 *pInputDeviceIDs
   );
   SLresult (*QueryAudioInputCapabilities)(
         SLAudioIODeviceCapabilitiesItf self,
         SLuint32 deviceID,
         SLAudioInputDescriptor *pDescriptor
   );
   SLresult (*RegisterAvailableAudioInputsChangedCallback) (
         SLAudioIODeviceCapabilitiesItf self,
         slAvailableAudioInputsChangedCallback callback,
         void *pContext
   );
   SLresult (*GetAvailableAudioOutputs)(
         SLAudioIODeviceCapabilitiesItf self,
         SLuint32 *pNumOutputs,
         SLuint32 *pOutputDeviceIDs
   );
   SLresult (*QueryAudioOutputCapabilities)(
         SLAudioIODeviceCapabilitiesItf self,
         SLuint32 deviceID,
         SLAudioOutputDescriptor *pDescriptor
   );
```

```
      SLresult (*RegisterAvailableAudioOutputsChangedCallback) (
            SLAudioIODeviceCapabilitiesItf self,
            slAvailableAudioOutputsChangedCallback callback,
            void *pContext
      );
      SLresult (*RegisterDefaultDeviceIDMapChangedCallback) (
            SLAudioIODeviceCapabilitiesItf self,
            slDefaultDeviceIDMapChangedCallback callback,
            void *pContext
            );
      SLresult (*GetAssociatedAudioInputs) (
            SLAudioIODeviceCapabilitiesItf self,
            SLuint32 deviceID,
            SLuint32 *pNumAudioInputs,
            SLuint32 *pAudioInputDeviceIDs
      );
      SLresult (*GetAssociatedAudioOutputs) (
            SLAudioIODeviceCapabilitiesItf self,
            SLuint32 deviceID,
            SLuint32 *pNumAudioOutputs,
            SLuint32 *pAudioOutputDeviceIDs
      );
      SLresult (*GetDefaultAudioDevices) (
            SLAudioIODeviceCapabilitiesItf self,
            SLuint32 defaultDeviceID,
            SLuint32 *pNumAudioDevices,
            SLuint32 *pAudioDeviceIDs
      );
      SLresult (*QuerySampleFormatsSupported)(
            SLAudioIODeviceCapabilitiesItf self,
            SLuint32 deviceID,
            SLmilliHertz samplingRate,
            SLint32 *pSampleFormats,
            SLuint32 *pNumOfSampleFormats
      );
   };
```

Interface ID

20e25b70-d02f-11df-bd3b-0800200c9a66

Defaults

I/O device capabilities vary widely from system to system. Defaults are not applicable.

Callbacks

slAvailableAudioInputsChangedCallback

```
typedef void (SLAPIENTRY *slAvailableAudioInputsChangedCallback) (
    SLAudioIODeviceCapabilitiesItf caller,
    void *pContext,
    SLuint32 deviceID,
    SLuint32 numInputs,
    SLboolean isNew
);
```

Description	This callback executes when the set of available audio input devices changes (as when a new Bluetooth headset is connected or a wired microphone is disconnected).		
Parameters	caller	[in]	The interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	deviceID	[in]	ID of the audio input device that has changed (that is, was either removed or added).
	numInputs	[in]	Updated number of available audio input devices.
	isNew	[in]	Set to SL_BOOLEAN_TRUE if the change was an addition of a newly available audio input device; SL_BOOLEAN_FALSE if an existing audio input device is no longer available.
Comments	The callback does not provide additional detail about the audio input device that has changed. In the case of an addition, it is up to the application to use QueryAudioInputCapabilities() to determine the full characteristics of the newly available audio input device.		
See Also	QueryAudioInputCapabilities()		

slAvailableAudioOutputsChangedCallback

```
typedef void (SLAPIENTRY *slAvailableAudioOutputsChangedCallback) (
   SLAudioIODeviceCapabilitiesItf caller,
   void *pContext,
   SLuint32 deviceID,
   SLuint32 numOutputs,
   SLboolean isNew
);
```

Description	This callback executes when the set of available audio output devices changes (as when a new Bluetooth headset is connected or a wired headset is disconnected).		
Parameters	caller	[in]	The interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	deviceID	[in]	ID of the audio output device that has changed (that is, was either removed or added).
	numOutputs	[in]	Updated number of available audio output devices.
	isNew	[in]	Set to SL_BOOLEAN_TRUE if the change was an addition of a newly available audio output device; SL_BOOLEAN_FALSE if an existing audio output device is no longer available.
Comments	The callback does not provide additional details about the audio output device that has changed. In the case of an addition, it is up to the application to use QueryAudioOutputCapabilities() to determine the full characteristics of the newly-available audio output device.		
See Also	QueryAudioOutputCapabilities()		

slDefaultDeviceIDMapChangedCallback

```
typedef void (SLAPIENTRY *slDefaultDeviceIDMapChangedCallback) (
   SLAudioIODeviceCapabilitiesItf caller,
   void *pContext,
   SLboolean isOutput,
   SLuint32 numDevices
);
```

Description	This callback executes when the set of audio output devices mapped to SL_DEFAULTDEVICEID_AUDIOINPUT or SL_DEFAULTDEVICEID_AUDIOOUTPUT changes		
Parameters	caller	[in]	The interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	isOutput	[in]	If true, then devices mapped to SL_DEFAULTDEVICEID_AUDIOOUTPUT have changed, otherwise the devices mapped to SL_DEFAULTDEVICEID_AUDIOINPUT have changed.
	numDevices	[in]	New number of physical audio output devices to which SL_DEFAULTDEVICEID_AUDIOOUTPUT or SL_DEFAULTDEVICEID_AUDIOINPUT is now mapped (depending on value of isOutput). Is always greater than or equal to 1.
Comments	The callback does not provide additional details about the audio devices now mapped to the default device ID. It is up to the application to retrieve the device IDs and to use the device IDs to query the capabilities of each device. numDevices is included in the callback for the benefit of those applications who may not wish to send/receive their audio stream to/from more than one device. Such applications can examine numDevices and opt to stop operation immediately if it is greater than 1, without needing to invoke other methods to get the new number of devices mapped to SL_DEFAULTDEVICEID_AUDIOOUTPUT or SL_DEFAULTDEVICEID_AUDIOINPUT.		
See Also	QueryAudioOutputCapabilities()		

Methods

GetAvailableAudioInputs

```
SLresult (*GetAvailableAudioInputs)(
    SLAudioIODeviceCapabilitiesItf self,
    SLuint32 *pNumInputs,
    SLuint32 *pInputDeviceIDs
);
```

Description	Gets the number and IDs of audio input devices currently available.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pNumInputs	[in/out]	As an input, specifies the length of the pInputDeviceIDs array (ignored if pInputDeviceIDs is NULL). As an output, specifies the number of audio input device IDs available in the system. Returns 0 if no audio input devices are available in the system.
	pInputDeviceIDs	[out]	Array of audio input device IDs currently available in the system. This parameter is populated by the call with the array of input device IDs (provided that pNumInputs is equal to or greater than the number of actual input device IDs). If pNumInputs is less than the number of actual input device IDs, the error code SL_RESULT_BUFFER_INSUFFICIENT is returned.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_BUFFER_INSUFFICIENT SL_RESULT_PARAMETER_INVALID		
Comments	Note that "available" implies those audio input devices that are active (that is, can accept input audio) and this number may be less than or equal to the total number of audio input devices in the system. For example, if a system has both an integrated microphone and a line-in jack, but the line-in jack is not connected to anything, the number of available audio inputs is only 1. Device IDs should not be expected to be contiguous. Device IDs are unique: the same device ID must not be used for different device types.		
See Also	GetAvailableAudioOutputs()		

QueryAudioInputCapabilities

```
SLresult (*QueryAudioInputCapabilities)(
    SLAudioIODeviceCapabilitiesItf self,
    SLuint32 deviceID,
    SLAudioInputDescriptor *pDescriptor
);
```

Description	Gets the capabilities of the specified audio input device.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	deviceID	[in]	ID of the audio input device.
	pDescriptor	[out]	Structure defining the capabilities of the audio input device [see section 9.1.3].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_IO_ERROR		
Comments	This method outputs a structure that contains one or more pointers to arrays If any pointers are NULL, the corresponding size field of the array will be the size needed to allocate the array.		
See Also	QueryAudioOutputCapabilities(), QuerySampleFormatsSupported()		

RegisterAvailableAudioInputsChangedCallback

```
SLresult (*RegisterAvailableAudioInputsChangedCallback)(
    SLAudioIODeviceCapabilitiesItf self,
    slAvailableAudioInputsChangedCallback callback,
    void *pContext
);
```

Description	Sets or clears `slAvailableAudioInputsChangedCallback()`.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`callback`	[in]	Address of the callback.
	`pContext`	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: `SL_RESULT_SUCCESS`. `SL_RESULT_PARAMETER_INVALID`		
Comments	None.		
See Also	`slAvailableAudioInputsChangedCallback()`		

GetAvailableAudioOutputs

```
SLresult (*GetAvailableAudioOutputs)(
   SLAudioIODeviceCapabilitiesItf self,
   SLuint32 *pNumOutputs,
  SLuint32 *pOutputDeviceIDs
);
```

Description	Gets the number and IDs of audio output devices currently available.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pNumOutputs`	[in/out]	As an input, specifies the size of the `pOutputDeviceIDs` array . As an output, specifies the number of audio output devices currently available in the system. Returns 0 if no audio output devices are active in the system.
	`pOutputDeviceIDs`	[out]	Array of audio output device IDs that are currently available in the system. This parameter is populated by the call with the array of output device IDs (provided that `pNumOutputs` is equal to or greater than the number of actual device IDs).
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_BUFFER_INSUFFICIENT SL_RESULT_PARAMETER_INVALID		
Comments	Note that “available” implies those audio output devices that are active (that is, can render audio) and this number may be less than or equal to the total number of audio output devices on the system. For example, if a system has both an integrated loudspeaker and a 3.5mm headphone jack, but if the headphone jack is not connected to anything, the number of available audio outputs is only 1. Device IDs should not be expected to be contiguous. Device IDs are unique: the same device ID must not be used for different device types.		
See Also	**8.13 GetAvailableAudioInputs()**		

QueryAudioOutputCapabilities

```
SLresult (*QueryAudioOutputCapabilities)(
   SLAudioIODeviceCapabilitiesItf self,
   SLuint32 deviceID,
   SLAudioOutputDescriptor *pDescriptor
);
```

Description	Gets the capabilities of the specified audio output device.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	deviceID	[in]	ID of the audio output device.
	pDescriptor	[out]	Structure defining the characteristics of the audio output device [see section 9.1.4].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_IO_ERROR		
Comments	This method outputs a structure that contains one or more pointers to arrays. If any pointers are NULL, the corresponding size field of the array will be the size needed to allocate the array.		
See Also	QueryAudioInputCapabilities(), QuerySampleFormatsSupported()		

RegisterAvailableAudioOutputsChangedCallback

```
SLresult (*RegisterAvailableAudioOutputsChangedCallback)(
   SLAudioIODeviceCapabilitiesItf self,
   slAvailableAudioOutputsChangedCallback callback,
   void *pContext
);
```

Description	Sets or clears `slAvailableAudioOutputsChangedCallback()`.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`callback`	[in]	Address of the callback.
	`pContext`	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See Also	`slAvailableAudioOutputsChangedCallback()`		

RegisterDefaultDeviceIDMapChangedCallback

```
SLresult (*RegisterDefaultDeviceIDMapChangedCallback)(
   SLAudioIODeviceCapabilitiesItf self,
   slDefaultDeviceIDMapChangedCallback callback,
   void *pContext
);
```

Description	Sets or clears `slDefaultDeviceIDMapChangedCallback()`.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`callback`	[in]	Address of the callback.
	`pContext`	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See Also	`slDefaultDeviceIDMapChangedCallback()`		

GetAssociatedAudioInputs

```
SLresult (*GetAssociatedAudioInputs) (
    SLAudioIODeviceCapabilitiesItf self,
    SLuint32 deviceID,
    SLuint32 *pNumAudioInputs,
    SLuint32 *pAudioInputDeviceIDs
);
```

Description	This method returns an array of audio input devices physically associated with this audio I/O device.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`deviceID`	[in]	ID of the input or output device.
	`pNumAudioInputs`	[in/out]	As an input, specifies the length of the `pAudioInputDeviceIDs` array. As an output, specifies the number of audio input device IDs associated with `deviceID`. Returns zero if there is no such association.
	`pAudioInputDeviceIDs`	[out]	Array of audio input device IDs. Should be ignored if `pNumAudioInputs` is zero – that is, if there are no associated audio inputs. This parameter is populated by the call with the array of input device IDs (provided that `pNumInputs` is equal to or greater than the number of actual input device IDs).

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_BUFFER_INSUFFICIENT`
	`SL_RESULT_IO_ERROR`		

GetAssociatedAudioInputs	
Comments	This method can be called on both audio input and audio output devices. It is useful for determining coupling of audio inputs and outputs on certain types of accessories. For example, it is helpful to know that microphone 01 is actually part of the same Bluetooth headset as speaker 03. Also, many car kits have multiple speakers and multiple microphones. Hence the need for an array of associated input devices. For applications that both accept and render audio, this method helps to determine whether an audio input and an audio output belong to the same physical accessory. An audio device cannot be associated with itself. So, in the example above, if this method were to be called with microphone 01 as the deviceID parameter, it would return an empty array, since there are no other inputs associated with microphone 01 on that Bluetooth headset.
See also	`GetDefaultAudioDevices()`

GetAssociatedAudioOutputs			
`SLresult (*GetAssociatedAudioOutputs) ( SLAudioIODeviceCapabilitiesItf self, SLuint32 deviceID, SLuint32 *pNumAudioOutputs, SLuint32 *pAudioOutputDeviceIDs );`			
Description	This method returns an array of audio output devices physically associated with this audio I/O device.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	deviceID	[in]	ID of the input or output device.
	pNumAudioOutputs	[in/out]	As an input, specifies the length of the pAudioOutputDeviceIDs array. As an output, specifies the number of audio output device IDs associated with deviceID. Returns zero if there is no such association.
	pAudioOutputDeviceIDs	[out]	Array of audio output device IDs. Should be ignored if pNumAudioOutputs is zero (that is, there are no associated audio outputs). This parameter is populated by the call with the array of output device IDs (provided that pNumAudioOutputs is equal to or greater than the number of actual device IDs).
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_BUFFER_INSUFFICIENT SL_RESULT_IO_ERROR		

GetAssociatedAudioOutputs	
Comments	This method can be called on both audio input and audio output devices. It is useful for determining coupling of audio inputs and outputs on certain types of accessories. For example, it is helpful to know that microphone 01 is actually part of the same Bluetooth headset as speaker 03. Also, many car kits have multiple speakers and multiple microphones. Hence the need for an array of associated output devices. For applications that both accept and render audio, this method helps to determine whether an audio input and an audio output belong to the same physical accessory. An audio device cannot be associated with itself. So, in the example above, if this method were to be called with speaker 03 as the deviceID parameter, it would return an empty array, since there are no other outputs associated with speaker 03 on that Bluetooth headset.
See also	`GetDefaultAudioDevices()`

GetDefaultAudioDevices

```
SLresult (*GetDefaultAudioDevices) (
   SLAudioIODeviceCapabilitiesItf self,
   SLuint32 defaultDeviceID,
   SLuint32 *pNumAudioDevices,
   SLuint32 *pAudioDeviceIDs
);
```

Description	Gets the number of audio devices currently mapped to the given default device ID.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	defaultDeviceID	[in]	ID of the default device (currently defined as SL_DEFAULTDEVICEID_AUDIOOUTPUT and SL_DEFAULTDEVICEID_AUDIOINPUT [see section 9.2.13]).
	pNumAudioDevices	[in/out]	As an input, specifies the length of the pAudioDeviceIDs array. As an output, specifies the number of audio device IDs mapped to the given defaultDeviceID.
	pAudioDeviceIDs	[out]	Array of audio device IDs that are currently mapped to the given defaultDeviceID. This parameter is populated by the call with the array of device IDs (provided that pNumAudioDevices is equal to or greater than the number of actual device IDs). If pNumAudioDevices is less than the number of actual mapped device IDs, the error code SL_RESULT_BUFFER_INSUFFICIENT is returned.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_BUFFER_INSUFFICIENT SL_RESULT_IO_ERROR SL_RESULT_PARAMETER_INVALID		
Comments	The mapping of defaultDeviceID to the physical audio devices (represented by the device IDs) is implementation-dependent. The application can choose to be notified of the implementation-induced changes to this mapping by registering for the slDefaultDeviceIDMapChangedCallback().		

GetDefaultAudioDevices	
See Also	`RegisterDefaultDeviceIDMapChangedCallback(), GetAssociatedAudioInputs(), GetAssociatedAudioOutputs()`

QuerySampleFormatsSupported			
`SLresult (*QuerySampleFormatsSupported) ( SLAudioIODeviceCapabilitiesItf self, SLuint32 deviceID, SLmilliHertz samplingRate, SLint32 *pSampleFormats, SLuint32 *pNumOfSampleFormats, );`			
Description	Gets an array of sample formats supported by the audio I/O device for the given sampling rate. The rationale here is that an audio I/O device might not support all sample formats at all sampling rates. Therefore, it is necessary to query the sample formats supported for each sampling rate of interest.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`deviceID`	[in]	ID of the audio I/O device
	`samplingRate`	[in]	Sampling rate for which the sampling formats are to be determined.
	`pSampleFormats`	[out]	Array of sample formats supported, as defined in the `SL_PCMSAMPLEFORMAT` macros. This parameter is populated by the call with the array of supported sample formats (provided that `pNumOfSampleFormats` is equal to or greater than the number of actual sample formats).
	`pNumOfSampleFormats`	[in/out]	As an input, specifies the length of the `pSampleFormats` array. As an output, specifies the number of sample formats supported.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_BUFFER_INSUFFICIENT`
	`SL_RESULT_IO_ERROR`		

Comments	None.		
See Also	`QueryAudioInputCapabilities()`, `QueryAudioOutputCapabilities()`		

8.13 SLBassBoostItf

Description

This interface is for controlling bass boost functionality.

This interface affects different parts of the audio processing chain, depending on which object the interface is exposed. If this interface is exposed on an Output Mix object, the effect is applied to the output mix. If this interface is exposed on a Player object, it is applied to the Player's output only. For more information, see section 4.5.1.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_BASSBOOST;

struct SLBassBoostItf_;
typedef const struct SLBassBoostItf_ * const * SLBassBoostItf;

struct SLBassBoostItf_ {
   SLresult (*SetEnabled)(
         SLBassBoostItf self,
         SLboolean enabled
   );
   SLresult (*IsEnabled)(
         SLBassBoostItf self,
         SLboolean *pEnabled
   );
   SLresult (*SetStrength)(
         SLBassBoostItf self,
         SLpermille strength
   );
   SLresult (*GetRoundedStrength)(
         SLBassBoostItf self,
         SLpermille *pStrength
   );
   SLresult (*IsStrengthSupported)(
         SLBassBoostItf self,
         SLboolean *pSupported
   );
};
```

Interface ID

0634f220-ddd4-11db-a0fc-0002a5d5c51b

Defaults

Enabled: false (disabled)

Strength: implementation-dependent

Methods

SetEnabled			
`SLresult (*SetEnabled)( SLBassBoostItf self, SLboolean enabled );`			
Description	Enables the effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	enabled	[in]	True to turn on the effect, false to switch it off.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	None.		

IsEnabled			
`SLresult (*IsEnabled)( SLBassBoostItf self, SLboolean *pEnabled );`			
Description	Gets the enabled status of the effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pEnabled	[out]	True if the effect is on, false otherwise.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetStrength

```
SLresult (*SetStrength)(
    SLBassBoostItf self,
    SLpermille strength
);
```

Description	Sets the strength of the bass boost effect. If the implementation does not support per mille accuracy for setting the strength, it is allowed to round the given strength to the nearest supported value. You can use the `GetRoundedStrength()` method to query the (possibly rounded) value that was actually set.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`strength`	[in]	Strength of the effect. The valid range for strength is [0, 1000], where 0 per mille designates the mildest effect and 1000 per mille designates the strongest.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_CONTROL_LOST`		

Comments	Please note that the strength does not affect the output if the effect is not enabled. This set method will in any event store the setting, even when the effect is not enabled currently. Please note also that the strength can also change if the output device is changed (as, for example, from loudspeakers to headphones) and those output devices use different algorithms with different accuracies. You can use device changed callbacks [see section 8.12] to monitor device changes and then query the possibly changed strength using `GetRoundedStrength()` if you want, for example, the graphical user interface to follow the current strength accurately.		

GetRoundedStrength			

`SLresult (*GetRoundedStrength)(`
`   SLBassBoostItf self,`
`   SLpermille *pStrength`
`);`			

Description	Gets the current strength of the effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pStrength	[out]	The strength of the effect. The valid range for strength is [0, 1000], where 0 per mille designates the mildest effect and 1000 per mille the strongest,
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Please note that the strength does not affect the output if the effect is not enabled. Please note also that in some cases the exact mapping of the strength to the underlying algorithms might not maintain the full accuracy exposed by the API.		

IsStrengthSupported			

`SLresult (*IsStrengthSupported)(`
`   SLBassBoostItf self,`
`   SLboolean *pSupported`
`);`			

Description	Indicates whether setting strength is supported. If this method returns false, only one strength is supported and the SetStrength method always rounds to that value.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pSupported	[out]	True if setting of the strength is supported, false if only one strength is supported.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.14 SLBufferQueueItf

Description

This interface is used for streaming audio data. It provides a method for queuing up buffers on a player object for playback by the device or on a recorder object for filling a queue of buffers. It also provides for a callback function called whenever a buffer in the queue is dequeued. The buffers are played or filled in the order in which they are queued. The order that they are dequeued is also the same as the order they are enqueued. The state of the buffer queue can be queried to provide information on the playback status of the buffer queue. This interface implements a simple streaming mechanism.

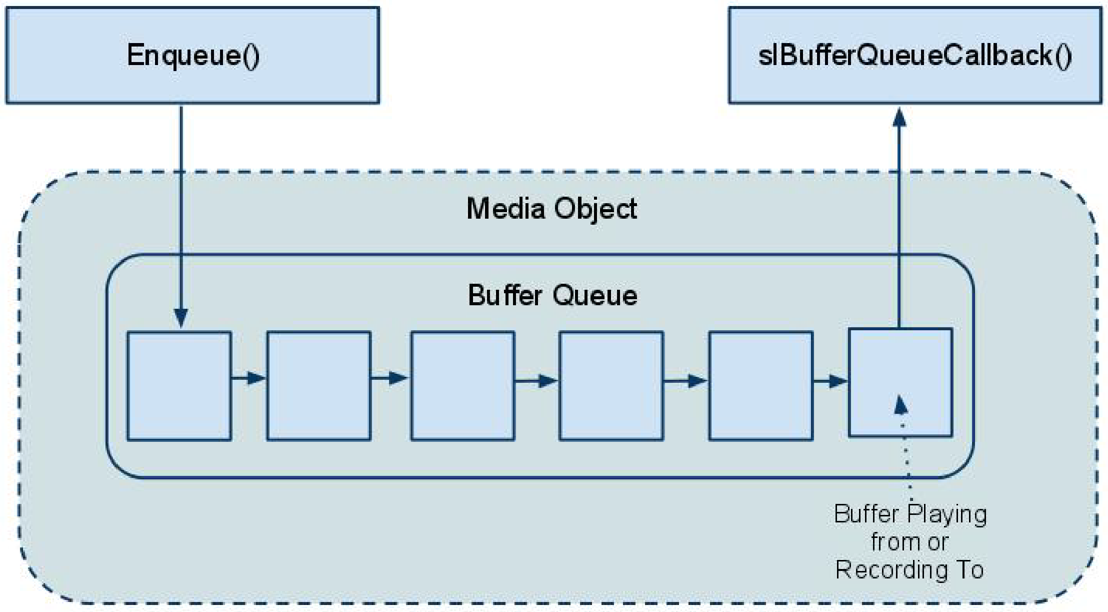


Figure 33: Overview of a buffer queue

- An attempt to instantiate an `SLBufferQueueItf` on a player object whose data source is not of type `SL_DATALOCATOR_BUFFERQUEUE` or `SL_DATALOCATOR_MIDIBUFFERQUEUE` is invalid and will fail.
- An attempt to instantiate an `SLBufferQueueItf` on a recorder object whose data sink is not of type `SL_DATALOCATOR_BUFFERQUEUE` is invalid and will fail.
- For player objects, when the player is in the `SL_PLAYSTATE_PLAYING` state, which is controlled by the `SLPlayItf` interface [see section 8.37], adding buffers will implicitly start playback. In the case of starvation due to insufficient buffers in the queue, the playing of audio data stops. The player remains in the `SL_PLAYSTATE_PLAYING` state.

Upon queuing of additional buffers, the playing of audio data resumes. Note that starvation of queued buffers causes audible gaps in the audio data stream. In the case where the player is not in the playing state, addition of buffers does not start audio playback.

- For recorder objects, when the recorder is in the SL_RECORDSTATE_RECORDING, which is controlled by the SLRecordItf interface [see section 8.42] adding buffers will implicitly start the filling process. In the case of starvation due to insufficient buffers in the queue, the filling of audio data stops and the audio data to be recorded into the buffer queue is lost for the period of the starvation. The recorder remains in the SL_RECORDSTATE_RECORDING state. Upon queuing of additional buffers, the filling of audio data resumes with the current audio data and not from where it left off when the starvation began. In the case where the recorder is not in the recording state, addition of buffers does not start the filling of any buffers in the queue.
- The buffers that are queued in a player object are used in place and are not required to be copied by the device, although this may be implementation-dependent. The application developer should be aware that modifying the content of a buffer after it has been queued is undefined and can cause audio corruption.
- Once an enqueued buffer has finished playing or filling, as notified by the callback notification, it is safe to delete the buffer. It is also safe to fill the buffer with new data and once again enqueue the buffer for playback on a player object or consume the buffer and then enqueue the buffer to be filled again on a recorder object.
- On transition to the SL_PLAYSTATE_STOPPED state the queue is cleared by releasing all buffers and the cursor is reset to 0. Callbacks will be called for each buffer released with the SL_BUFFERQUEUEEVENT_STOPPED event flag set.
- On transition to the SL_RECORDSTATE_STOPPED state, the application should continue to enqueue buffers onto the queue to retrieve the residual recorded data in the system. The retrieval of residual recorded data is complete when the application receives the buffer queue callback notification with the SL_BUFFERQUEUEEVENT_CONTENT_END event flag set. Any empty buffers in the queue will remain as-is, for possible re-use for the next recording session. The record cursor is reset to 0.
- On transition to the SL_PLAYSTATE_PAUSED or SL_RECORDSTATE_PAUSED state the cursor remains at the current position in the buffer.

Figure 34 shows what SL_BUFFERQUEUEEVENT flags will be set and what audio data will be played for a given buffer queue on a player object.

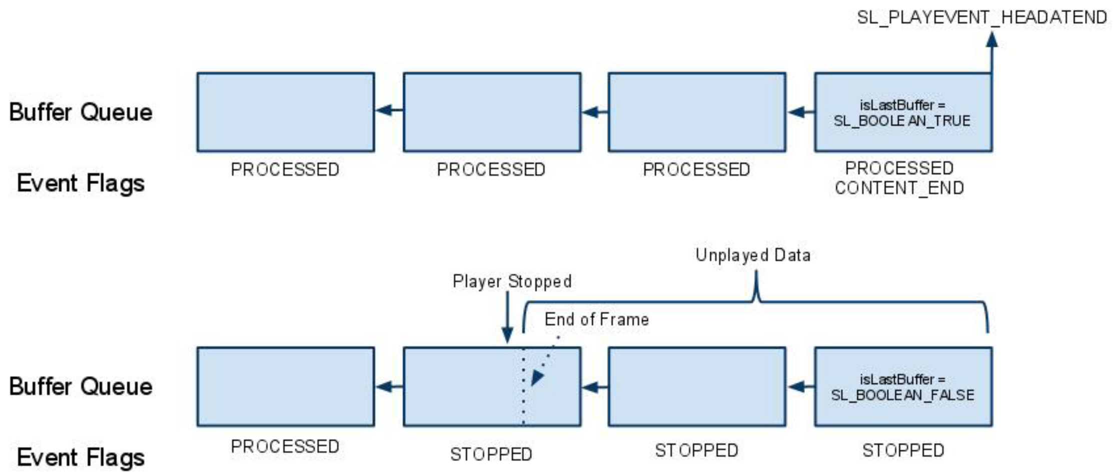


Figure 34: Buffer queue event flags.

This interface is supported on the Audio Player [see section 7.2] object object and the Audio Recorder [see section 7.3] object.

See sections B.1.1 and C.4.2 for examples using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_BUFFERQUEUE;

struct SLBufferQueueItf_;
typedef const struct SLBufferQueueItf_ * const * SLBufferQueueItf;

struct SLBufferQueueItf_ {
   SLresult (*Enqueue) (
         SLBufferQueueItf self,
         const void *pBuffer,
         SLuint32 size,
         SLboolean isLastBuffer
   );
   SLresult (*Clear) (
         SLBufferQueueItf self
   );
   SLresult (*GetState) (
         SLBufferQueueItf self,
         SLBufferQueueState *pState
   );
   SLresult (*RegisterCallback) (
         SLBufferQueueItf self,
         slBufferQueueCallback callback,
            void* pContext
      );
      SLresult (*SetCallbackEventsMask) (
            SLBufferQueueItf self,
            SLuint32 eventFlags
      );
      SLresult (*GetCallbackEventsMask) (
            SLBufferQueueItf self,
            SLuint32 *pEventFlags
      );
};
```

Interface ID

524b68a0-d3d6-11df-bd3b-0800200c9a66

Defaults

- No buffers are queued.
- No callback method is registered.
- Event mask is 0.

Callbacks

slBufferQueueCallback

```
typedef void (SLAPIENTRY *slBufferQueueCallback) (
   SLBufferQueueItf caller,
   SLuint32 eventFlags,
   const void *pBuffer,
   SLuint32 bufferSize,
   SLuint32 dataUsed,
   void *pContext
);
```

Description	Callback function called when a buffer is dequeued from the buffer queue.		
Parameters	caller	[in]	Interface instantiation on which the callback was registered.
	eventFlags	[in]	Combination of buffer queue event flags. See SL_BUFFERQUEUEEVENT.
	pBuffer	[in]	A pointer to the buffer dequeued.
	bufferSize	[in]	Size of pBuffer as specified at the time of the buffer was enqueued.
	dataUsed	[in]	Size of data used in pBuffer.
	pContext	[in]	User context data that is supplied when the callback method is registered.
Comments	Note that the size of the data used in the buffer may be less than the		

slBufferQueueCallback	
	size of the buffer.
See Also	`RegisterCallback()`

Methods

Enqueue

```
SLresult (*Enqueue) (
   SLBufferQueueItf self,
   const void *pBuffer,
   SLuint32 size,
   SLboolean isLastBuffer
);
```

Description	Adds a buffer to the queue. The method takes a pointer to the data to queue and the size in bytes of the buffer as arguments. The buffers are played or filled in the order in which they are queued using this method.		
Pre-conditions	The buffer queue does not have a buffer with isLastBuffer set to SL_BOOLEAN_TRUE.		
Parameters	self	[in]	Interface self-reference.
	pBuffer	[in]	Pointer to the buffer data to enqueue. Must be non-NULL.
	size	[in]	Size of data in bytes. Must be greater than zero.
	isLastBuffer	[in]	Set to SL_BOOLEAN_TRUE if this is the last buffer of data enqueued.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_BUFFER_INSUFFICIENT SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	For a player object, when the object is in the SL_PLAYSTATE_PLAYING state, which is controlled by the SLPlayItf interface [see section 8.37], adding buffers will implicitly start playback. In the case of starvation due to insufficient buffers in the queue the playing of audio data stops. The media object remains in the SL_PLAYSTATE_PLAYING state. Upon queuing of additional buffers, the playing of audio data resumes. Note that starvation of queued buffers causes audible gaps in the audio data stream. If the maximum number of buffers specified in the SLDataLocator_BufferQueue structure used as the data source when creating the media object using the CreateAudioPlayer or CreateMidiPlayer method has been reached, the buffer is not added to the buffer queue and SL_RESULT_BUFFER_INSUFFICIENT is returned. At this point the client should wait until it receives a callback notification for a buffer completion at which time it can enqueue the buffer. For a recorder object, when the object is in the SL_RECORDSTATE_RECORDING state, which is controlled by the SLRecordItf interface [see section 8.37] adding buffers will implicitly start the filling process. In the case of starvation due to insufficient		

Enqueue	
	buffers in the queue, the filling of buffers will stop and the audio data to be recorded into the buffer queue is lost for the period of the starvation. The media object remains in the `SL_RECORDSTATE_RECORDING` state. Upon queuing of additional buffers, the filling of audio data resumes with the current audio data and not from where it left off when the starvation began. If the maximum number of buffers specified in the `SLDataLocator_BufferQueue` structure used as the data sink when creating the recorder object using the `CreateAudioRecorder` method has been reached, the buffer is not added to the buffer queue and `SL_RESULT_BUFFER_INSUFFICIENT` is returned. At this point the client should wait until it receives a callback notification for a buffer completion at which time it can enqueue the buffer. The `isLastBuffer` flag is useful when buffer queues are used as data sources for playback. It is set to `SL_BOOLEAN_TRUE` for the last buffer of data enqueued by the application. This notifies the implementation that this is indeed the last buffer of data that needs to be played out the system. The implementation will know to wait until this last buffer of data is played out the audio output device before sending the SL_PLAYEVENT_HEADATEND event on SLPLayItf. The `isLastBuffer` flag is ignored when buffer queues are used as data sinks for recording.

Clear			
`SLresult (*Clear)( SLBufferQueueItf self );`			
Description	Releases all buffers currently queued in the buffer queue. The `SLBufferQueueState` is reset to the initial state.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS		
Comments	This method resets the cumulative position information used in the `SLPlayItf` interfaces for `GetPosition()`.Buffers dequeued from this call will have the SL_BUFFERQUEUEEVENT_CLEARED event flag set in `slBufferQueueCallback`.		

GetState			
`SLresult (*GetState)( SLBufferQueueItf self, SLBufferQueueState *pState );`			
Description	Returns the current state of the buffer queue.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pState`	[out]	Pointer to a location ready to receive the buffer queue state. The `SLBufferQueueState` structure contains information on the current number of queued buffers and the index of the currently playing or filling buffer.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

RegisterCallback

```
SLresult (*RegisterCallback)(
   SLBufferQueueItf self,
   slBufferQueueCallback callback,
   void *pContext
);
```

Description	Sets the callback function to be called on buffer completion.		
Pre-conditions	The RegisterCallback() method can only be called while the media object is in the SL_PLAYSTATE_STOPPED or SL_RECORDSTATE_STOPPED state.		
Parameters	self	[in]	Interface self-reference.
	callback	[in]	Pointer to callback function to call on buffer completion. The callback is called on the completion of each buffer in the queue. If NULL, the callback is disabled.
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	The RegisterCallback() method can only be called while the media object is in the SL_PLAYSTATE_STOPPED or SL_RECORDSTATE_STOPPED state, to avoid race conditions between removal of the callback function and callbacks that may be in process. The callback function may be changed by calling the RegisterCallback() method multiple times, but only while the media object is in the stopped state. SL_RESULT_PRECONDITIONS_VIOLATED is returned if the media object is not in the SL_PLAYSTATE_STOPPED state when the method is called.		

SetCallbackEventsMask

```
SLresult (*SetCallbackEventsMask) (
   SLBufferQueueItf self,
   SLuint32 eventFlags
);
```

Description	Enables/disables notification of buffer queue events.		
Pre-conditions	None.		
Parameters	Self	[in]	Interface self-reference.
	eventFlags	[in]	Bitmask of buffer queue event flags (see SL_BUFFERQUEUEEVENT macros in section 0) indicating which callback events are enabled. The presence of a flag enables notification for the corresponding event. The absence of a flag disables notification for the corresponding event.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The callback event flags default to all flags cleared.		

GetCallbackEventsMask

```
SLresult (*GetCallbackEventsMask) (
   SLBufferQueueItf self,
   SLuint32 *pEventFlags
);
```

Description	Queries for the notification state (enabled/disabled) of buffer queue events.		
Pre-conditions	None.		
Parameters	Self	[in]	Interface self-reference.
	pEventFlags	[out]	Pointer to a location to receive the bitmask of buffer queue event flags (see SL_BUFFERQUEUEEVENT macros in section 0) indicating which callback events are enabled. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.15 SLConfigExtensionsItf

Description

This interface provides a mechanism for an application to set and query the configuration of the underlying audio engine. These configuration parameters are in the form of key-value pairs. As such, the method signatures do not assume any vendor-specific or platform-specific knowledge of the underlying audio components. The methods of this interface have been designed such that they can be used to get/set the parameters for any OpenSL ES object in a vendor-specific manner. Therefore, the usage of this interface is not limited to the audio engine or codecs. It applicable to all OpenSL ES objects.

This interface can be exposed on any OpenSL ES object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_CONFIGEXTENSION;

struct SLConfigExtensionsItf_;
typedef const struct SLConfigExtensionsItf_
    * const * SLConfigExtensionsItf;

struct SLConfigExtensionsItf_ {
    SLresult (*SetConfiguration) (
        SLConfigExtensionsItf self,
        const SLchar * pConfigKey,
        SLuint32 valueSize,
        const void * pConfigValue
    );
    SLresult (*GetConfiguration) (
        SLConfigExtensionsItf self,
        const SLchar * pConfigKey,
        SLuint32 * pValueSize,
        void * pConfigValue
    );
};
```

Interface ID

151d35b0-c375-11df-851a-0800200c9a66

Methods

SetConfiguration			
`SLresult (*SetConfiguration) ( SLConfigExtensionsItf self, const SLchar * pConfigKey, SLuint32 valueSize; const void * pConfigValue );`			
Description	Sets the configuration as a key-value pair		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pConfigKey	[in]	String representing the “key” – the parameter/attribute name of the configuration.
	valueSize	[in]	The size of the value referenced by pConfigValue, in bytes.
	pConfigValue	[in]	Address of the parameter/attribute being set.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_READONLY SL_RESULT_PARAMETER_INVALID		
Comments	The configValue input parameter is passed by reference. For example, this method could be used to specify an obscure codec parameter for use during an encoding operation, or force extended HTTP headers into a URI request for a cloud-based audio file. It is up to the underlying object to appropriately parse the key-value pair and make sense of the parameter setting.		

GetConfiguration

```
SLresult (*GetConfiguration) (
    SLConfigExtensionItf self,
    const SLchar * pConfigKey,
    SLuint32 * pValueSize,
    void * pConfigValue
);
```

Description	Gets the configuration setting as a key-value pair		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pConfigKey	[in]	String representing the "key" – the name of the parameter/attribute being queried. If configKey is not recognized as a valid parameter/attributes of the underlying object SL_RESULT_PARAMETER_INVALID is return.
	pValueSize	[in/out]	Address of the size of the memory block passed as pConfigValue.
	pConfigValue	[out]	Address of the value of the parameter/attribute that is returned. If the size of the memory block passed as pConfigValue is too small to return the entire value, SL_RESULT_PARAMETER_INVALID is returned.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	If the memory area specified by pConfigValue and pValueSize is too small to receive the entire value, only the first pValueSize bytes will be returned in pConfigValue. pValueSize will be set to the minimum size required for the call to succeed. The pConfigValue output parameter is passed by reference. For example, this method could be used for querying the RTSP proxy IP address and port number (123.213.123.5:80), or the bearer-specific bandwidth limits (900-1800 MHz). It is up to the underlying object to appropriately parse the key string and return the corresponding parameter setting, in the appropriate format. An error is returned if the key is not recognized by the underlying object.		

8.16 SLDeviceVolumeItf

Description

This interface exposes controls for manipulating the volume of specific audio input and audio output devices. The units used for setting and getting the volume can be in millibels or as arbitrary volume steps; the units supported by the device can be queried with GetVolumeScale method.

Support for this interface is optional, but, where supported, this interface should be exposed on the engine object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_DEVICEVOLUME;

struct SLDeviceVolumeItf_;
typedef const struct SLDeviceVolumeItf_ * const * SLDeviceVolumeItf;

struct SLDeviceVolumeItf_ {
   SLresult (*GetVolumeScale) (
         SLDeviceVolumeItf self,
         SLuint32 deviceID,
         SLint32 *pMinValue,
         SLint32 *pMaxValue,
         SLboolean *pIsMillibelScale
   );
   SLresult (*SetVolume) (
         SLDeviceVolumeItf self,
         SLuint32 deviceID,
         SLint32 volume
   );
   SLresult (*GetVolume) (
         SLDeviceVolumeItf self,
         SLuint32 deviceID,
         SLint32 *pVolume
   );
};
```

Interface ID

e1634760-f3e2-11db-9ca9-0002a5d5c51b

Defaults

The default volume setting of each device should be audible.

Methods

GetVolumeScale			
`SLresult (*GetVolumeScale) ( SLDeviceVolumeItf self, SLuint32 deviceID, SLint32 *pMinValue, SLint32 *pMaxValue, SLboolean *pIsMillibelScale );`			
Description	Gets the properties of the volume scale supported by the given device.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	deviceID	[in]	Audio input or output device's identifier.
	pMinValue	[out]	The smallest supported volume value of the device.
	pMaxValue	[out]	The greatest supported volume value of the device.
	pIsMillibelScale	[out]	If true, the volume values used by GetVolume, SetVolume and this method are in millibel units; if false, the volume values are in arbitrary volume steps.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	This method may return SL_RESULT_FEATURE_UNSUPPORTED if the specified device does not support changes to its volume. The scale is always continuous and device-specific. It could be, for example, [0, 15] if arbitrary volume steps are used or [-32768, 0] if millibels are used.		
See also	SLAudioIODeviceCapabilitiesItf(), SLOutputMixItf()		

SetVolume

```
SLresult (*SetVolume) (
   SLDeviceVolumeItf self,
   SLuint32 deviceID,
   SLint32 volume
);
```

Description	Sets the device's volume level.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	deviceID	[in]	Device's identifier.
	volume	[in]	The new volume setting. The valid range is continuous and its boundaries can be queried from GetVolumeScale method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	The minimum and maximum supported volumes are device-dependent. This method may fail if the specified device does not support changes to its volume or the volume is outside the range supported by the device.		
See also	SLAudioIODeviceCapabilitiesItf(), SLOutputMixItf()		

GetVolume			
`SLresult (*GetVolume) ( SLDeviceVolumeItf self, SLuint32 deviceID, SLint32 *pVolume );`			
Description	Gets the device's volume.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	deviceID	[in]	Device's identifier.
	pVolume	[out]	Pointer to a location to receive the object's volume setting. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	SL_RESULT_FEATURE_UNSUPPORTED is returned if the specified device does not support changes to its volume.		

8.17 SLDynamicInterfaceManagementItf

Description

The `SLDynamicInterfaceManagementItf` interface provides methods for handling interface exposure on an object after the creation and realization of the object. The primary method for exposing interfaces on an object is by listing them in the engine object's creation methods [see section 8.21].

`SLDynamicInterfaceManagementItf` is an implicit interface of all object types. Please refer to section 3.1.6 for details about how dynamically exposed interfaces work with the object states and other exposed interfaces.

This interface is supported on all objects [see section 7].

See sections B.2 and C.2 for examples using this interface.

Defaults

No dynamic interfaces are exposed.

No callback is registered.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_DYNAMICINTERFACEMANAGEMENT;

struct SLDynamicInterfaceManagementItf_;
typedef const struct SLDynamicInterfaceManagementItf_
         * const * SLDynamicInterfaceManagementItf;

struct SLDynamicInterfaceManagementItf_ {
   SLresult (*AddInterface) (
         SLDynamicInterfaceManagementItf self,
         const SLInterfaceID iid,
         SLboolean async
   );
   SLresult (*RemoveInterface) (
         SLDynamicInterfaceManagementItf self,
         const SLInterfaceID iid
   );
   SLresult (*ResumeInterface) (
         SLDynamicInterfaceManagementItf self,
         const SLInterfaceID iid,
         SLboolean async
   );
   SLresult (*RegisterCallback) (
         SLDynamicInterfaceManagementItf self,
         slDynamicInterfaceManagementCallback callback,
         void * pContext
   );
};
```

Interface ID

63936540-f775-11db-9cc4-0002a5d5c51b

Callbacks

slDynamicInterfaceManagementCallback

```
typedef void (SLAPIENTRY *slDynamicInterfaceManagementCallback) (
   SLDynamicInterfaceManagementItf caller,
   void * pContext,
   SLuint32 event,
   SLresult result,
   const SLInterfaceID iid
);
```

Description	A callback function, notifying of a runtime error, termination of an asynchronous call or change in a dynamic interface's resources.		
Parameters	caller	[in]	Interface that invoked the callback.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	event	[in]	One of the Dynamic Interface Management Event macros.
	result	[in]	Contains either the error code, if the event is SL_DYNAMIC_ITF_EVENT_RUNTIME_ERROR, or the asynchronous function return code, if the event is SL_DYNAMIC_ITF_EVENT_ASYNC_TERMINATION. The result may be: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE
	iid	[in]	Interface type ID that the event affects.
Comments	Please note the restrictions applying to operations performed from within callback context in section 3.3.		
See also	RegisterCallback()		

Methods

AddInterface

```
SLresult (*AddInterface) (
    SLDynamicInterfaceManagementItf self,
    const SLInterfaceID iid,
    SLboolean async
);
```

Description	Optionally asynchronous method for exposing an interface on an object. In asynchronous mode the success or failure of exposing the interface will be sent to the registered callback function.		
Pre-conditions	Interface has not been exposed.		
Parameters	self	[in]	Interface self-reference.
	iid	[in]	Valid interface type ID.
	async	[in]	If SL_BOOLEAN_FALSE, the method will block until termination. Otherwise, the method will return SL_RESULT_SUCCESS, and will be executed asynchronously. However, if the implementation is unable to initiate the asynchronous call SL_RESULT_RESOURCE_ERROR will be returned.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_RESOURCE_ERROR SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	When successful, the interface is exposed on the object and the interface pointer can be obtained by SLObjectItf::GetInterface(). Adding the interface to the object acquires the resources required for its functionality. The operation may fail if insufficient resources are available. In such a case, the application may wait until resources become available (SL_DYNAMIC_ITF_EVENT_RESOURCES_AVAILABLE event is sent [see section 9.2.17]), and then resume the interface. Additionally, the application may increase the object's priority, thus increasing the likelihood that the object will steal another object's resources. Adding an interface that is already exposed will result in a return value of SL_RESULT_PRECONDITIONS_VIOLATED.		
See also	SLObjectItf::GetInterface()		

RemoveInterface

```
SLresult (*RemoveInterface) (
   SLDynamicInterfaceManagementItf self,
   const SLInterfaceID iid
);
```

Description	Synchronous method for removing a dynamically exposed interface on the object. This method is supported in all object states.		
Pre-conditions	Interface has been exposed.		
Parameters	`self`	[in]	Interface self-reference.
	`iid`	[in]	Valid interface type ID that has been exposed on this object by use of the `AddInterface()` method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	An object that is suspended or unrealized waits also for resources for dynamically managed interfaces before sending a SL_DYNAMIC_ITF_EVENT_RESOURCES_AVAILABLE event [see section 9.2.17]. By removing a dynamic interface in an Unrealized or Suspended state, the object does not wait for resources for that dynamic interface. Removing an interface that is not exposed will result in a return value of SL_RESULT_PRECONDITIONS_VIOLATED.		
See also	None.		

ResumeInterface

```
SLresult (*ResumeInterface) (
    SLDynamicInterfaceManagementItf self,
    const SLInterfaceID iid,
    SLboolean async
);
```

Description	Optionally asynchronous method for resuming a dynamically exposed interface on the object.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	iid	[in]	Valid interface type ID that has been exposed on this object by use of the AddInterface() method.
	async	[in]	If SL_BOOLEAN_FALSE, the method will block until termination. Otherwise, the method will return SL_RESULT_SUCCESS, and will be executed asynchronously. However, if the implementation is unable to initiate the asynchronous call SL_RESULT_RESOURCE_ERROR will be returned.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	When successful, the interface is exposed on the object and the interface pointer can be obtained by SLObjectItf::GetInterface(). This method can be used on a suspended dynamic interface after reception of a SL_DYNAMIC_ITF_EVENT_RESOURCES_AVAILABLE event [see section 9.2.17].		
See also	None.		

RegisterCallback

```
SLresult (*RegisterCallback) (
   SLDynamicInterfaceManagementItf self,
   slDynamicInterfaceManagementCallback callback,
   void * pContext
);
```

Description	Registers a callback on the object that executes when a runtime error, termination of an asynchronous call or change in a dynamic interface's resources occurs.		
Parameters	self	[in]	Interface self-reference.
	callback	[in]	Address of the result callback. If NULL, the callback is disabled.
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS		
Comments	None.		
See also	slDynamicInterfaceManagementCallback()		

8.18 SLDynamicSourceItf

Description

This interface is deprecated. Use SLDynamicSourceSinkChangeItf [see section 8.19] instead.

This interface exposes a control for changing the data source of the object during the life-time of the object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_DYNAMICSOURCE;

struct SLDynamicSourceItf_;
typedef const struct SLDynamicSourceItf_ * const * SLDynamicSourceItf;

struct SLDynamicSourceItf_ {
   SLresult (*SetSource) (
         SLDynamicSourceItf self,
         const SLDataSource *pDataSource
   );
};
```

Interface ID

01e0dbe0-d032-11df-bd3b-0800200c9a66

Defaults

The data source set by application during object creation.

Methods

SetSource			
`SLresult (*SetSource) ( SLDynamicSourceItf self, const SLDataSource *pDataSource );`			
Description	Sets the data source for the object.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference
	pDataSource	[in]	Pointer to the structure specifying the media data source (such as a container file). Must be non-NULL. In the case of a Metadata Extractor object, only local data sources are mandated to be supported.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR SL_RESULT_CONTENT_CORRUPTED SL_RESULT_CONTENT_UNSUPPORTED SL_RESULT_CONTENT_NOT_FOUND SL_RESULT_PERMISSION_DENIED		
Comments	Setting a source for a Metadata Extractor object will reset its SLMetadataExtractionItf and SLMetadataTraversalItf interfaces [see section 8.26 and 0] to point to the new source and reset those interfaces to their initial values. Setting of the new source shall be accepted in any player object state. The playback of the new source shall start from the beginning of the content. The player object shall maintain the same player object state upon accepting the new source. For example, if the player object is currently in SL_PLAYSTATE_PLAYING state, it shall maintain the SL_PLAYSTATE_PLAYING state.		
See also	None.		

8.19 SLDynamicSourceSinkChangeItf

Description

This interface exposes a control for changing the specified data source or data sink of the object during the life-time of the object. This interface is explicitly optional on the AudioPlayer, MIDI Player and AudioRecorder objects and implicitly mandated on the Metadata Extractor object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_DYNAMICSOURCESINKCHANGE;

struct SLDynamicSourceSinkChangeItf_;
typedef const struct SLDynamicSourceSinkChangeItf_ * const *
SLDynamicSourceSinkChangeItf;

struct SLDynamicSourceSinkChangeItf_ {
    SLresult (*ChangeSource) (
        SLDynamicSourceSinkChangeItf self,
        const SLDataSource * pExistingDataSource,
        const SLDataSource * pNewDataSource,
        SLboolean async
    );
    SLresult (*ChangeSink) (
        SLDynamicSourceSinkChangeItf self,
        const SLDataSink * pExistingDataSink,
        const SLDataSink * pNewDataSink,
        SLboolean async
    );
    SLresult (*RegisterSourceChangeCallback) (
        SLDynamicSourceSinkChangeItf self,
        slSourceChangeCallback callback,
        void * pContext
    );
    SLresult (*RegisterSinkChangeCallback) (
        SLDynamicSourceSinkChangeItf self,
        slSinkChangeCallback callback,
        void * pContext
    );
};
```

Interface ID

c3b4c270-c5da-11df-bd3b-0800200c9a66

Defaults

None.

Callbacks

slSourceChangeCallback			
`typedef void (SLAPIENTRY *slSourceChangeCallback)( SLDynamicSourceSinkChangeItf caller, void *pContext, SLuint32 resultCode, const SLDataSource *pExistingDataSource, const SLDataSource *pNewDataSource );`			
Description	Callback that executes whenever the ChangeSource method finishes execution.		
Parameters	caller	[in]	Interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	resultCode	[in]	Result code indicating the status of the source change. Possible result codes are the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR SL_RESULT_CONTENT_CORRUPTED SL_RESULT_CONTENT_UNSUPPORTED SL_RESULT_CONTENT_NOT_FOUND SL_RESULT_PERMISSION_DENIED SL_RESULT_FORMATS_INCOMPATIBLE
	pExistingDataSource	[in]	Pointer to the current data source passed in the ChangeSource method.
	pNewDataSource	[in]	Pointer to the new data source passed in the ChangeSource method.

slSourceChangeCallback	
Comments	The resultCode indicates whether the source change requested by ChangeSource actually succeeded or not. SL_RESULT_SOURCE_SINK_INCOMPATIBLE is returned if the application tries to connect two media objects with incompatible media formats and/or locator types. SL_RESULT_PERMISSION_DENIED is returned if the source to be connected has protected content and the media object does not have the requisite permissions to process it. The pExistingDataSource and pNewDataSource pointers are exactly the same pointers passed in the ChangeSource method in the SLDynamicSourceSinkChangeItf. As such, no implementation memory space is being exposed by this callback. It is worth noting that: • Two successive calls to ChangeSource asking for the same change would result in only one callback. That is, the second call to ChangeSource is ignored. • Two successive calls to ChangeSource asking for different changes will result in two callbacks in the correct order.
See Also	RegisterSourceChangeCallback, ChangeSource

slSinkChangeCallback

```
typedef void (SLAPIENTRY *slSinkChangeCallback)(
      SLDynamicSourceSinkChangeItf caller,
      void *pContext,
      SLuint32 resultCode,
      const SLDataSource *pExistingDataSink,
      const SLDataSource *pNewDataSink
);
```

Description	Callback that executes whenever the ChangeSink method finishes execution.		
Parameters	caller	[in]	Interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	resultCode	[in]	Result code indicating the status of the source change. Possible result codes are the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR SL_RESULT_CONTENT_CORRUPTED SL_RESULT_CONTENT_UNSUPPORTED SL_RESULT_CONTENT_NOT_FOUND SL_RESULT_PERMISSION_DENIED SL_RESULT_FORMATS_INCOMPATIBLE

slSinkChangeCallback			
	pExistingDataSink	[in]	Pointer to the current data sink passed in the ChangeSink method.
	pNewDataSink	[in]	Pointer to the new data sink passed in the ChangeSink method.
Comments	The resultCode indicates whether the Sink change requested by ChangeSink actually succeeded or not. `SL_RESULT_SOURCE_SINK_INCOMPATIBLE` is returned if the application tries to connect two media objects with incompatible media formats. `SL_RESULT_PERMISSION_DENIED` is returned if the sink to be connected does not have the requisite permissions to handle the protected media content output by the media object. The `pExistingDataSink` and `pNewDataSink` pointers are exactly the same pointers passed in the ChangeSink method in the `SLDynamicSourceSinkChangeItf`. As such, no implementation memory space is being exposed by this callback. It is worth noting that: • Two successive calls to ChangeSink asking for the same change would result in only one callback. That is, the second call to ChangeSink is ignored. • Two successive calls to ChangeSink asking for different changes will result in two callbacks in the correct order.		
See Also	`RegisterSinkChangeCallback, ChangeSink`		

Methods

ChangeSource			

`SLresult (*ChangeSource) (`
`SLDynamicSourceSinkChangeItf self,`
`const SLDataSource *pExistingDataSource,`
`const SLDataSource *pNewDataSource,`
`SLboolean async`
`);`			

Description	Changes the specified data source of the audio object.		
Parameters	self	[in]	Interface self-reference.
	pExistingDataSource	[in]	Pointer to the structure specifying the existing media data source. Must be non-NULL.
	pNewDatSource	[in]	Pointer to the structure specifying the new media data source. Must be non-NULL. In the case of a Metadata Extractor object, only local data sources are mandated to be supported.

ChangeSource			
	async	[in]	If SL_BOOLEAN_FALSE, the method will block until termination. Otherwise, the method will return SL_RESULT_SUCCESS, and will be executed asynchronously. On termination, the `slSourceChangeCallback()` will be invoked, if registered, The `resultCode` parameter of the `callback` will contain the result code of the method. However, if the implementation is unable to initiate the asynchronous call SL_RESULT_RESOURCE_ERROR will be returned.

Return Value	The following return values are possible.
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_MEMORY_FAILURE`
	`SL_RESULT_IO_ERROR`
	`SL_RESULT_CONTENT_CORRUPTED`
	`SL_RESULT_CONTENT_UNSUPPORTED`
	`SL_RESULT_CONTENT_NOT_FOUND`
	`SL_RESULT_PERMISSION_DENIED`
	`SL_RESULT_FORMATS_INCOMPATIBLE`
	
	If invoked as an asynchronous method, it returns SL_RESULT_SUCCESS, and the resultCode parameter of the callback will contain the result of this method's execution.		

Comments	Setting of the new source shall be accepted in any media object state. • For audio player objects, playback of the new source will start from the beginning of the content. • For audio recorder objects, the recording of the new source will start from the beginning of the content. The media object shall maintain the same media object state upon accepting the new source. For example, if the player object is currently in SL_PLAYSTATE_PLAYING state, it shall maintain the SL_PLAYSTATE_PLAYING state. This method is asynchronous. The application would need to register for the `slSourceChangeCallback` to get notification of the outcome of the source change.		
See Also	`slSourceChangeCallback, RegisterSourceChangeCallback`		

ChangeSink

```
SLresult (*ChangeSink) (
      SLDynamicSourceSinkChangeItf self,
      const SLDataSink *pExistingDataSink,
      const SLDataSink *pNewDataSink,
      SLboolean async
);
```

Description	Changes the specified data sink of the audio object.		
Parameters	self	[in]	Interface self-reference.
	pExistingDataSink	[in]	Pointer to the structure specifying the existing media data sink. Must be non-NULL.
	pNewDatSink	[in]	Pointer to the structure specifying the new media data sink. Must be non-NULL.
	async	[in]	If SL_BOOLEAN_FALSE, the method will block until termination. Otherwise, the method will return SL_RESULT_SUCCESS, and will be executed asynchronously. On termination, the slSinkChangeCallback() will be invoked, if registered, The resultCode parameter of the callback will contain the result code of the method. However, if the implementation is unable to initiate the asynchronous call SL_RESULT_RESOURCE_ERROR will be returned.
Return Value	The following return values are possible. SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR SL_RESULT_CONTENT_CORRUPTED SL_RESULT_CONTENT_UNSUPPORTED SL_RESULT_CONTENT_NOT_FOUND SL_RESULT_PERMISSION_DENIED SL_RESULT_FORMATS_INCOMPATIBLE If invoked as an asynchronous method, it returns SL_RESULT_SUCCESS, and the resultCode parameter of the callback will contain the result of this method's execution.		

ChangeSink	
Comments	Setting of the new sink shall be accepted in any media object state. • For audio player objects, the playback of the content to the new sink will continue from the current position in the content. • For audio recorder objects, the recording of the content to the new sink will continue from the current position in the content. The media object shall maintain the same media object state upon accepting the new sink. For example, if the player object is currently in `SL_PLAYSTATE_PLAYING` state, it shall maintain the `SL_PLAYSTATE_PLAYING` state. This method is asynchronous. The application would need to register for the slSinkChangeCallback to get notification of the outcome of the sink change.
See Also	slSinkChangeCallback, RegisterSinkChangeCallback

RegisterSourceChangeCallback

```
    SLresult (*RegisterSourceChangeCallback) (
          SLDynamicSourceSinkChangeItf self,
          slSourceChangeCallback callback,
          void *pContext
    );
```

Description	Sets or clears slSourceChangeCallback.		
Parameters	self	[in]	Interface self-reference.
	callback	[in]	Pointer to the callback function to be called when the ChangeSource method finishes execution.
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return Value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See Also	slSourceChangeCallback		

RegisterSinkChangeCallback

```
    SLresult (*RegisterSinkChangeCallback) (
          SLDynamicSourceSinkChangeItf self,
          slSinkChangeCallback callback,
          void *pContext
    );
```

Description	Sets or clears slSinkChangeCallback.		
Parameters	Self	[in]	Interface self-reference.
	Callback	[in]	Pointer to the callback function to be called when the ChangeSink method finishes execution.

RegisterSinkChangeCallback			
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return Value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See Also	slSinkChangeCallback		

8.20 SLEffectSendItf

Description

This interface allows an application developer to control a sound's contribution to auxiliary effects. An auxiliary effect is identified by its interface pointer. The only auxiliary effect standardized in the specification is a single reverb. Some implementations may expose other auxiliary effects.

The auxiliary effect will need to be enabled for this interface to have any audible effect. For the reverb effect, this requires exposing either PresetReverb [see section 8.40] or EnvironmentalReverb [see Section 8.23] on the Output Mix object.

This interface is supported on the Audio Player [see section 7.2] and MIDI Player [see section 7.8] objects.

See sections B.6.1, C.2 and C.4 for examples using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_EFFECTSEND;

struct SLEffectSendItf_;
typedef const struct SLEffectSendItf_ * const * SLEffectSendItf;

struct SLEffectSendItf_ {
   SLresult (*EnableEffectSend) (
         SLEffectSendItf self,
         const void *pAuxEffect,
         SLboolean enable,
         SLmillibel initialLevel
   );
   SLresult (*IsEnabled) (
         SLEffectSendItf self,
         const void * pAuxEffect,
         SLboolean *pEnable
   );
   SLresult (*SetDirectLevel) (
         SLEffectSendItf self,
         SLmillibel directLevel
   );
   SLresult (*GetDirectLevel) (
         SLEffectSendItf self,
         SLmillibel *pDirectLevel
   );
   SLresult (*SetSendLevel) (
         SLEffectSendItf self,
         const void *pAuxEffect,
         SLmillibel sendLevel
   );
   SLresult (*GetSendLevel)(
         SLEffectSendItf self,
         const void *pAuxEffect,
         SLmillibel *pSendLevel
   );
};
```

Interface ID

56e7d200-ddd4-11db-aefb-0002a5d5c51b

Defaults

Direct level: 0 mB (no level change)

No effect sends enabled.

Methods

EnableEffectSend

```
SLresult (*EnableEffectSend) (
   SLEffectSendItf self,
   const void *pAuxEffect,
   SLboolean enable,
   SLmillibel initialLevel
);
```

Description	Enables or disables the player's contribution to an auxiliary effect.		
Pre-conditions	None		
Parameters	self	[in]	Interface self-reference.
	pAuxEffect	[in]	The auxiliary effect's interface.
	enable	[in]	If true, the path to the auxiliary effect is enabled. If false, the path to the auxiliary effect is disabled.
	initialLevel	[in]	Player's send path level for the specified effect. The valid range is [SL_MILLIBEL_MIN, 0].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE		
Comments	This method may fail with the return value SL_RESULT_RESOURCE_ERROR, if the implementation is unable to feed the player's output to the specified effect. The implementation shall return SL_RESULT_PARAMETER_INVALID if pAuxEffect is not a valid effect interface for use with the effect send interface.		

IsEnabled			
```
SLresult (*IsEnabled) (
   SLEffectSendItf self,
   const void *pAuxEffect,
   SLboolean *pEnable
);
```
Description	Returns whether a player's output is fed into an auxiliary effect.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pAuxEffect`	[in]	Pointer to the auxiliary effect's interface.
	`pEnable`	[out]	Pointer to a location for the enabled status. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	The implementation shall return `SL_RESULT_PARAMETER_INVALID` if `pAuxEffect` is not a valid effect interface for use with the effect send interface.		

SetDirectLevel			
```
SLresult (*SetDirectLevel) (
   SLEffectSendItf self,
   SLmillibel directLevel
);
```
Description	Sets the dry (direct) path level for a sound.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`directLevel`	[in]	Direct path level in millibels. The valid range is [`SL_MILLIBEL_MIN`, 0].

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

GetDirectLevel

```
SLresult (*GetDirectLevel) (
   SLEffectSendItf self,
   SLmillibel *pDirectLevel
);
```

Description	Gets the player's direct path level.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pDirectLevel`	[out]	Pointer to a location for receiving the direct path level in millibels.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetSendLevel

```
SLresult (*SetSendLevel) (
   SLEffectSendItf self,
   const void *pAuxEffect,
   SLmillibel sendLevel
);
```

Description	Sets the player's send path level for a specified auxiliary effect.		
Pre-conditions	Effect send is enabled (using `EnableEffectSend()`).		
Parameters	`self`	[in]	Interface self-reference.
	`pAuxEffect`	[in]	Pointer to the auxiliary effect's interface.
	`sendLevel`	[in]	Send level in millibels. The valid range is [`SL_MILLIBEL_MIN`, `0`].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	None.		

GetSendLevel			
`SLresult (*GetSendLevel) ( SLEffectSendItf self, const void *pAuxEffect, SLmillibel *pSendLevel );`			
Description	Gets the player’s send path level for a specified auxiliary effect.		
Pre-conditions	Effect send is enabled (using `EnableEffectSend()`).		
Parameters	`self`	[in]	Interface self-reference.
	`pAuxEffect`	[in]	Pointer to the auxiliary effect’s interface.
	`pSendLevel`	[out]	Pointer to location for receiving the send level in millibels.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_PRECONDITIONS_VIOLATED`		

Comments	None.		

8.21 SLEngineItf

Description

This interface exposes creation methods of all the OpenSL ES object types.

This interface is supported on the engine [see section 7.4] object.

See Appendix B: and Appendix C: for examples using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_ENGINE;

struct SLEngineItf_;
typedef const struct SLEngineItf_ * const * SLEngineItf;

struct SLEngineItf_ {
   SLresult (*CreateLEDDevice) (
         SLEngineItf self,
         SLObjectItf * pDevice,
         SLuint32 deviceID,
         SLuint32 numInterfaces,
         const SLInterfaceID * pInterfaceIds,
         const SLboolean * pInterfaceRequired
   );
   SLresult (*CreateVibraDevice) (
         SLEngineItf self,
         SLObjectItf * pDevice,
         SLuint32 deviceID,
         SLuint32 numInterfaces,
         const SLInterfaceID * pInterfaceIds,
         const SLboolean * pInterfaceRequired
   );
   SLresult (*CreateAudioPlayer) (
         SLEngineItf self,
         SLObjectItf * pPlayer,
         const SLDataSource *pAudioSrc,
         const SLDataSink *pAudioSnk,
         SLuint32 numInterfaces,
         const SLInterfaceID * pInterfaceIds,
         const SLboolean * pInterfaceRequired
   );
```

```
SLresult (*CreateAudioRecorder) (
      SLEngineItf self,
      SLObjectItf * pRecorder,
      const SLDataSource *pAudioSrc,
      const SLDataSink *pAudioSnk,
      SLuint32 numInterfaces,
      const SLInterfaceID * pInterfaceIds,
      const SLboolean * pInterfaceRequired
);
SLresult (*CreateMidiPlayer) (
      SLEngineItf self,
      SLObjectItf * pPlayer,
      const SLDataSource *pMIDISrc,
      const SLDataSource *pBankSrc,
      const SLDataSink *pAudioOutput,
      const SLDataSink *pVibra,
      const SLDataSink *pLEDArray,
      SLuint32 numInterfaces,
      const SLInterfaceID * pInterfaceIds,
      const SLboolean * pInterfaceRequired
);
SLresult (*CreateListener) (
      SLEngineItf self,
      SLObjectItf * pListener,
      SLuint32 numInterfaces,
      const SLInterfaceID * pInterfaceIds,
      const SLboolean * pInterfaceRequired
);
SLresult (*Create3DGroup) (
      SLEngineItf self,
      SLObjectItf * pGroup,
      SLuint32 numInterfaces,
      const SLInterfaceID * pInterfaceIds,
      const SLboolean * pInterfaceRequired
);
SLresult (*CreateOutputMix) (
      SLEngineItf self,
      SLObjectItf * pMix,
      SLuint32 numInterfaces,
      const SLInterfaceID * pInterfaceIds,
      const SLboolean * pInterfaceRequired
);
SLresult (*CreateMetadataExtractor) (
      SLEngineItf self,
      SLObjectItf * pMetadataExtractor,
      const SLDataSource * pDataSource,
      SLuint32 numInterfaces,
      const SLInterfaceID * pInterfaceIds,
      const SLboolean * pInterfaceRequired
);
```

```
      SLresult (*CreateExtensionObject) (
            SLEngineItf self,
            SLObjectItf * pObject,
            void * pParameters,
            SLuint32 objectID,
            SLuint32 numInterfaces,
            const SLInterfaceID * pInterfaceIds,
            const SLboolean * pInterfaceRequired
      );
      SLresult (*QueryNumSupportedInterfaces) (
            SLEngineItf self,
            SLuint32 objectID,
            SLuint32 * pNumSupportedInterfaces
      );
      SLresult (*QuerySupportedInterfaces) (
            SLEngineItf self,
            SLuint32 objectID,
            SLuint32 index,
            SLInterfaceID * pInterfaceId
      );
      SLresult (*QueryNumSupportedExtensions) (
            SLEngineItf self,
            SLuint32 * pNumExtensions
      );
      SLresult (*QuerySupportedExtension) (
            SLEngineItf self,
            SLuint32 index,
            SLchar * pExtensionName,
            SLuint16 * pNameLength
      );
      SLresult (*IsExtensionSupported) (
            SLEngineItf self,
            const SLchar * pExtensionName,
            SLboolean * pSupported
      );
   };
```

Interface ID

5d1eb540-d032-11df-bd3b-0800200c9a66

Defaults

None (the interface is stateless)

Methods

CreateLEDDevice

```
SLresult (*CreateLEDDevice) (
    SLEngineItf self,
    SLObjectItf * pDevice,
    SLuint32 deviceID,
    SLuint32 numInterfaces,
    const SLInterfaceID * pInterfaceIds,
    const SLboolean * pInterfaceRequired
);
```

Description	Creates an LED device.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pDevice`	[out]	Newly-created LED device object.
	`deviceID`	[in]	ID of the LED device.
	`numInterfaces`	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	`pInterfaceIds`	[in]	Array of `numInterfaces` interface IDs, which the object should support. This parameter is ignored if `numInterfaces` is zero.
	`pInterfaceRequired`	[in]	Array of `numInterfaces` flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if `numInterfaces` is zero.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR		
Comments	None.		
See also	`SLEngineCapabilitiesItf` [section 8.22] to determine the capabilities of the LED device. LED Device Object [section 7.5]		

CreateVibraDevice

```
SLresult (*CreateVibraDevice) (
   SLEngineItf self,
   SLObjectItf * pDevice,
   SLuint32 deviceID,
   SLuint32 numInterfaces,
   const SLInterfaceID * pInterfaceIds,
   const SLboolean * pInterfaceRequired
);
```

Description	Creates a vibrator device.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pDevice	[out]	Newly-created vibrator device object.
	deviceID	[in]	ID of the vibrator device.
	numInterfaces	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	pInterfaceIds	[in]	Array of numInterfaces interface IDs, which the object should support. This parameter is ignored if numInterfaces is zero.
	pInterfaceRequired	[in]	Array of numInterfaces flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if numInterfaces is zero.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR		
Comments	None.		
See also	SLEngineCapabilitiesItf [section 8.22] to determine the capabilities of the vibrator device. Vibra Device Object [section 7.10]		

CreateAudioPlayer

```
SLresult (*CreateAudioPlayer) (
   SLEngineItf self,
   SLObjectItf * pPlayer,
   const SLDataSource *pAudioSrc,
   const SLDataSink *pAudioSnk,
   SLuint32 numInterfaces,
   const SLInterfaceID * pInterfaceIds,
   const SLboolean * pInterfaceRequired
);
```

Description	Creates an audio player object.		
Pre-conditions	If the data sink's locator is an object (e.g. output mix) this object must be in the realized state.		
Parameters	`self`	[in]	Interface self-reference.
	`pPlayer`	[out]	Newly created audio player object.
	`pAudioSrc`	[in]	Pointer to the structure specifying the audio data source (e.g. a compressed audio file). Must be non-`NULL`.
	`pAudioSnk`	[in]	Pointer to the structure specifying the audio data sink (such as the audio output mix).
	`numInterfaces`	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	`pInterfaceIds`	[in]	Array of `numInterfaces` interface IDs, which the object should support. This parameter is ignored if `numInterfaces` is zero.
	`pInterfaceRequired`	[in]	Array of `numInterfaces` flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if `numInterfaces` is zero.

CreateAudioPlayer	

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_MEMORY_FAILURE`
	`SL_RESULT_IO_ERROR`
	`SL_RESULT_CONTENT_CORRUPTED`
	`SL_RESULT_CONTENT_UNSUPPORTED`
	`SL_RESULT_CONTENT_NOT_FOUND`
	`SL_RESULT_PERMISSION_DENIED`

See also	Audio Player Object [section 7.2]

CreateAudioRecorder

```
SLresult (*CreateAudioRecorder) (
   SLEngineItf self,
   SLObjectItf * pRecorder,
   const SLDataSource *pAudioSrc,
   const SLDataSink *pAudioSnk,
   SLuint32 numInterfaces,
   const SLInterfaceID * pInterfaceIds,
   const SLboolean * pInterfaceRequired
);
```

Description	Creates an audio recorder.		
Pre-conditions	The data source is not a buffer queue.		
Parameters	`self`	[in]	Interface self-reference.
	`pRecorder`	[out]	Newly created audio recorder object.
	`pAudioSrc`	[in]	Pointer to the structure specifying the audio data source (such as a microphone device).
	`pAudioSnk`	[in]	Pointer to the structure specifying the audio data sink (such as a compressed audio output file). Must be non-`NULL`. If `pAudioSnk` is a URI (say a file) that already exists, it will be overwritten.
	`numInterfaces`	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	`pInterfaceIds`	[in]	Array of `numInterfaces` interface IDs, which the object should support. This parameter is ignored if `numInterfaces` is zero.
	`pInterfaceRequired`	[in]	Array of `numInterfaces` flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if `numInterfaces` is zero.

CreateAudioRecorder	

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_MEMORY_FAILURE`
	`SL_RESULT_IO_ERROR`
	`SL_RESULT_BUFFER_INSUFFICIENT`
	`SL_RESULT_CONTENT_UNSUPPORTED`
	`SL_RESULT_PERMISSION_DENIED`
	`SL_RESULT_READONLY`

See also	Audio Recorder Object [section 7.3]
Comments	A value of `SL_RESULT_READONLY` will be returned when `pAudioSnk` is unable to be written to due to read-only file status. SL_RESULT_PARAMETER_INVALID will be returned when the data source is a buffer queue.

CreateMidiPlayer

```
SLresult (*CreateMidiPlayer) (
   SLEngineItf self,
   SLObjectItf * pPlayer,
   const SLDataSource *pMIDISrc,
   const SLDataSource *pBankSrc,
   const SLDataSink *pAudioOutput,
   const SLDataSink *pVibra,
   const SLDataSink *pLEDArray,
   SLuint32 numInterfaces,
   const SLInterfaceID * pInterfaceIds,
   const SLboolean* pInterfaceRequired
);
```

Description	Creates a MIDI player.		
Pre-conditions	If the audio output's locator is an object (e.g. output mix) this object must be in the realized state.		
Parameters	`self`	[in]	Interface self-reference.
	`pPlayer`	[out]	Newly created MIDI player object.
	`pMIDISrc`	[in]	Pointer to the structure specifying the MIDI data source. This data source must be a Mobile XMF [mXMF] or SP-MIDI [SP-MIDI] file reference, or a MIDI buffer queue. This is an optional parameter. If `NULL`, no source is specified and MIDI events are created programmatically only.
	`pBankSrc`	[in]	Pointer to the structure specifying the instrument bank in Mobile DLS format. This is an optional parameter. If `NULL`, the default bank of instruments definitions is used.
	`pAudioOutput`	[in]	Pointer to the structure specifying the audio data sink (such as an audio output device).
	`pVibra`	[in]	Pointer to the structure specifying the vibra device to which the MIDI player should send vibra data. This is an optional parameter. If `NULL`, no vibra devices will be controlled by the MIDI data.

CreateMidiPlayer			
	`pLEDArray`	[in]	Pointer to the structure specifying the LED array device to which the MIDI player should send LED array data. This is an optional parameter. If `NULL`, no LED array devices will be controlled by the MIDI data.
	`numInterfaces`	[in]	Number of interface that the object is requested to support (not including implicit interfaces).
	`pInterfaceIds`	[in]	Array of `numInterfaces` interface IDs, which the object should support. This parameter is ignored if `numInterfaces` is zero.
	`pInterfaceRequired`	[in]	An array of `numInterfaces` flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if `numInterfaces` is zero.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_MEMORY_FAILURE`
	`SL_RESULT_IO_ERROR`
	`SL_RESULT_CONTENT_CORRUPTED`
	`SL_RESULT_CONTENT_UNSUPPORTED`
	`SL_RESULT_CONTENT_NOT_FOUND`
	`SL_RESULT_PERMISSION_DENIED`		

Comments	The player creation may fail with a `SL_RESULT_FEATURE_UNSUPPORTED` code if a bank is provided in the `pBankSrc` parameter and in addition a bank is embedded in the MIDI source. In such a case, the application is advised to try creating the MIDI player without providing a bank in `pBankSrc`. The application may wish to set `pMIDISrc` to `NULL` if it plans to solely use the MIDI messaging interface `SLMIDIMessageItf` [see section 8.29] to feed MIDI data from a single player to the MIDI synthesizer.		
See also	MIDI Player Object [section 7.8]		

CreateListener

```
SLresult (*CreateListener) (
    SLEngineItf self,
    SLObjectItf * pListener,
    SLuint32 numInterfaces,
    const SLInterfaceID * pInterfaceIds,
    const SLboolean* pInterfaceRequired
);
```

Description	Creates a listener.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pListener`	[out]	Newly created listener object.
	`numInterfaces`	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	`pInterfaceIds`	[in]	Array of `numInterfaces` interface IDs, which the object should support. This parameter is ignored if `numInterfaces` is zero.
	`pInterfaceRequired`	[in]	Array of `numInterfaces` flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if `numInterfaces` is zero.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_MEMORY_FAILURE`		

Comments	None.		
See also	Listener Object [section 7.6]		

Create3DGroup

```
SLresult (*Create3DGroup) (
   SLEngineItf self,
   SLObjectItf * pGroup,
   SLuint32 numInterfaces,
   const SLInterfaceID * pInterfaceIds,
   const SLboolean* pInterfaceRequired
);
```

Description	Creates a 3D group.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pGroup	[out]	Newly created 3D group object.
	numInterfaces	[in]	Number of interface that the object is requested to support (not including implicit interfaces).
	pInterfaceIds	[in]	Array of numInterfaces interface IDs, which the object should support. This parameter is ignored if numInterfaces is zero.
	pInterfaceRequired	[in]	Array of numInterfaces flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if numInterfaces is zero.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE		
Comments	None.		
See also	3D Group Object [section 7.1]		

CreateOutputMix

```
SLresult (*CreateOutputMix) (
    SLEngineItf self,
    SLObjectItf * pMix,
    SLuint32 numInterfaces,
    const SLInterfaceID * pInterfaceIds,
    const SLboolean * pInterfaceRequired
);
```

Description	Creates an output mix.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pMix`	[out]	Newly created output mix object.
	`numInterfaces`	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	`pInterfaceIds`	[in]	Array of `numInterfaces` interface IDs, which the object should support. This parameter is ignored if `numInterfaces` is zero.
	`pInterfaceRequired`	[in]	Array of `numInterfaces` flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if `numInterfaces` is zero.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE		
Comments	None.		
See also	Output Mix Object [section 7.9]		

CreateMetadataExtractor

```
SLresult (*CreateMetadataExtractor) (
   SLEngineItf self,
   SLObjectItf * pMetadataExtractor,
   SLDataSource * pDataSource,
   SLuint32 numInterfaces,
   const SLInterfaceID * pInterfaceIds,
   const SLboolean * pInterfaceRequired
);
```

Description	Creates a Metadata Extractor object.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMetadataExtractor	[out]	Newly created metadata extractor object.
	pDataSource	[in]	Pointer to the structure specifying the audio data source (such as a media file). Only local data sources are mandated to be supported. Must be non-NULL.
	numInterfaces	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	pInterfaceIds	[in]	Array of numInterfaces interface IDs, which the object should support. This parameter is ignored if numInterfaces is zero.
	pInterfaceRequired	[in]	Array of numInterfaces flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated. This parameter is ignored if numInterfaces is zero.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR SL_RESULT_CONTENT_CORRUPTED SL_RESULT_CONTENT_UNSUPPORTED SL_RESULT_CONTENT_NOT_FOUND SL_RESULT_PERMISSION_DENIED		

CreateMetadataExtractor	
Comments	None.
See also	Metadata Extractor Object [see section 7.7]

CreateExtensionObject

```
SLresult (*CreateExtensionObject) (
    SLEngineItf self,
    SLObjectItf * pObject,
    void * pParameters,
    SLuint32 objectID,
    SLuint32 numInterfaces,
    const SLInterfaceID * pInterfaceIds,
    const SLboolean * pInterfaceRequired
);
```

Description	Creates an object. This method is used for extension objects defined externally from the specification. Objects defined by the specification must be created by the specific creation methods in the engine interface.		
Pre-conditions	As documented by extension.		
Parameters	self	[in]	Interface self-reference.
	pObject	[out]	Newly-created object.
	pParameters	[in]	Pointer to a structure specifying the parameters used for creating the object.
	objectID	[in]	A valid object ID.
	numInterfaces	[in]	Number of interfaces that the object is requested to support (not including implicit interfaces).
	pInterfaceIds	[in]	Array of numInterfaces interface IDs, which the object should support.
	pInterfaceRequired	[in]	Array of numInterfaces flags, each specifying whether the respective interface is required on the object or optional. A required interface will fail the creation of the object if it cannot be accommodated.

CreateExtensionObject	
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PRECONDITIONS_VIOLATED SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR SL_RESULT_PERMISSION_DENIED
Comments	If the engine fails to create the object due to lack of memory or resources it will return the SL_RESULT_MEMORY_FAILURE or the SL_RESULT_RESOURCE_ERROR error, respectively. The ObjectID and the data structure pointed to by pParameters should be defined by an extension. When ObjectID is not valid the method will return SL_RESULT_FEATURE_UNSUPPORTED.
See also	Section 3.5

QueryNumSupportedInterfaces

```
SLresult (*QueryNumSupportedInterfaces) (
   SLEngineItf self,
   SLuint32 objectID,
   SLuint32 * pNumSupportedInterfaces
);
```

Description	Queries the number of supported interfaces available.		
Parameters	self	[in]	Interface self-reference.
	objectID	[in]	ID of the object being queried. Refer to SL_OBJECTID type. If the engine does not support the identified object this method will return SL_RESULT_FEATURE_UNSUPPORTED.
	pNumSupportedInterfaces	[out]	Identifies the number of supported interfaces available.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The number of supported interfaces will include both mandated and optional interfaces available for the object. This method can be used to determine whether or not an object is supported by an implementation by examining the return value.		

QueryNumSupportedInterfaces

See also	`QuerySupportedInterfaces()`, `slQueryNumSupportedEngineInterfaces()` [see section 6.2].

QuerySupportedInterfaces

```
SLresult (*QuerySupportedInterfaces) (
    SLEngineItf self,
    SLuint32 objectID,
    SLuint32 index,
    SLInterfaceID * pInterfaceId
);
```

Description	Queries the supported interfaces.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`objectID`	[in]	ID of the object being queried. Refer to SL_OBJECTID type. If the engine does not support the identified object this method will return SL_RESULT_FEATURE_UNSUPPORTED.
	`index`	[in]	Index used to enumerate available interfaces. Supported index range is 0 to N-1, where N is the number of supported interfaces.
	`pInterfaceId`	[out]	Identifies the supported interface.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The number of supported interfaces will include both mandated and optional interfaces available for the object.		
See also	`QueryNumberSupportedInterfaces()`, `slQueryNumSupportedEngineInterfaces()` [see section 6.3].		

QueryNumSupportedExtensions

```
SLresult (*QueryNumSupportedExtensions) (
    SLEngineItf self,
    SLuint32 * pNumExtensions
);
```

Description	Queries the number of supported extensions.		
Parameters	`self`	[in]	Interface self-reference.

QueryNumSupportedExtensions			
	pNumExtensions	[out]	Identifies the number of supported extensions by this implementation.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The number of supported extensions will include both standardized extensions listed in Khronos registry and vendor-specific extensions.		
See also	QuerySupportedExtensions()		

QuerySupportedExtension

```
SLresult (*QuerySupportedExtension) (
    SLEngineItf self,
    SLuint32 index,
    SLchar * pExtensionName,
    SLuint16 * pNameLength
);
```

Description	Gets the name of the extension supported by the implementation based on the given index.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	The index of the extension. Must be [0, numExtensions-1].
	pExtensionName	[out]	The name of the supported extension, as defined in the Khronos registry (http://www.khronos.org/registry/) or in vendor-specific documentation. The length of the needed char array should be first figured out from pNameLength out parameter by calling this method with pExtensionName as null.
	pNameLength	[in/out]	As an output, specifies the length of the name including the terminating NULL. As an input, specifies the length of the given pExtensionName char array (ignored if pExtensionName is NULL).

QuerySupportedExtension	
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_BUFFER_INSUFFICIENT
Comments	If the given length is smaller than the needed size SL_RESULT_BUFFER_INSUFFICIENT is returned and only data of the given size will be written; however, no invalid strings are written. That is, the null-terminator always exists and multibyte characters are not cut in the middle.
See Also	QueryNumSupportedExtensions(), IsExtensionSupported()

IsExtensionSupported

```
SLresult (*IsExtensionSupported) (
    SLEngineItf self,
    const SLchar * pExtensionName,
    SLboolean * pSupported
);
```

Description	Queries if the given extension is supported by the implementation.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pExtensionName	[in]	The name of an extension, as defined in the Khronos registry (http://www.khronos.org/registry/) or in vendor-specific documentation. Must be null-terminated.
	pSupported	[out]	SL_BOOLEAN_TRUE if the given extension is supported; SL_BOOLEAN_FALSE if it is not supported.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This is an alternative method to be used instead of QueryNumSupportedExtensions() and QuerySupportedExtension() to query the availability of just one known extension.		
See Also	None.		

8.22 SLEngineCapabilitiesItf

Description

Different implementations of OpenSL ES can support one or more of the three profiles (Phone, Music and Game). Further, these implementations can also vary in their ability to support simultaneous 2D and 3D sampled audio voices, as well as the MIDI polyphony level and the ability to treat the output of the MIDI synthesizer(s) as 3D sound source(s). For these reasons, an interface to query the capabilities of the OpenSL ES engine is necessary. The `SLEngineCapabilitiesItf` interface provides this functionality.

Version 1.1 of OpenSL ES mandates support for "at least one" MIDI synthesizer. This does not prevent implementations from supporting more than one synthesizer. Therefore, this interface allows querying of the number of MIDI synthesizers supported.

This interface is supported on the engine [see section 7.4] object.

See section B.7.1 for an example using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_ENGINECAPABILITIES;

struct SLEngineCapabilitiesItf_;
typedef const struct SLEngineCapabilitiesItf_* const *
SLEngineCapabilitiesItf;

struct SLEngineCapabilitiesItf_ {
   SLresult (*QuerySupportedProfiles) (
         SLEngineCapabilitiesItf self,
         SLuint16 *pProfilesSupported
   );
   SLresult (*QueryAvailableVoices) (
         SLEngineCapabilitiesItf self,
         SLuint16 voiceType,
         SLuint16 *pNumMaxVoices,
         SLboolean *pIsAbsoluteMax,
         SLuint16 *pNumFreeVoices
   );
   SLresult (*QueryNumberOfMIDISynthesizers) (
         SLEngineCapabilitiesItf self,
         SLuint16 *pNumMIDIsynthesizers
   );
```

```
      SLresult (*QueryAPIVersion) (
            SLEngineCapabilitiesItf self,
            SLuint16 *pMajor,
            SLuint16 *pMinor,
            SLuint16 *pStep
      );
      SLresult (*QueryLEDCapabilities) (
            SLEngineCapabilitiesItf self,
            SLuint32 *pIndex,
            SLuint32 *pLEDDeviceID,
            SLLEDDescriptor *pDescriptor
      );
      SLresult (*QueryVibraCapabilities) (
            SLEngineCapabilitiesItf self,
            SLuint32 *pIndex,
            SLuint32 *pVibraDeviceID,
            SLVibraDescriptor *pDescriptor
      );
      SLresult (*IsThreadSafe) (
            SLEngineCapabilitiesItf self,
            SLboolean *pIsThreadSafe
      );
   };
```

Interface ID

8320d0a0-ddd5-11db-a1b1-0002a5d5c51b

Defaults

None (the interface is stateless).

Methods

QuerySupportedProfiles			
`SLresult (*QuerySupportedProfiles)( SLEngineCapabilitiesIt self, SLuint16 *pProfilesSupported );`			
Description	Queries the supported profiles of the OpenSL ES API		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pProfilesSupported	[out]	Bitmask containing one or more of the three profiles supported, as defined in the SL_PROFILE macros.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See Also	Section 9.2.40 (SL_PROFILE macros)		

QueryAvailableVoices			

`SLresult (*QueryAvailableVoices)(`
`SLEngineCapabilitiesItf self,`
`SLuint16 voiceType,`
`SLuint16 *pNumMaxVoices,`
`SLboolean *pIsAbsoluteMax,`
`SLuint16 *pNumFreeVoices`
`);`			

Description	Queries the number of simultaneous free voices currently available.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	voiceType	[in]	One of the voice types listed in the SL_VOICETYPE macros.
	pNumMaxVoices	[out]	Maximum number of simultaneous voices of type “voiceType” supported by the implementation. For MIDI, it refers to the maximum polyphony level.
	pIsAbsoluteMax	[out]	SL_BOOLEAN_TRUE if the numMaxVoices returned is an absolute maximum that the device cannot exceed in any circumstances. This can be caused, for example, by hardware limitations in hardware-based implementations. SL_BOOLEAN_FALSE if the numMaxVoices returned specifies the maximum number of voices that the application is **recommended** to have active simultaneously (as might be recommended for a typical game playing situation when graphics rendering (2D or 3D) and reverberation are also taking place). The implementation does not guarantee that this number of voices may be active simultaneously under all conditions; it is just a hint to the application. (It may even be possible for more than this number of voices to be active simultaneously, but this is not guaranteed, either.) SL_BOOLEAN_FALSE is typically returned in software-based implementations.

QueryAvailableVoices			
	pNumFreeVoices	[out]	Number of voices of type “voiceType” currently available. For MIDI, it refers to the currently available polyphony level. Typically, “numFreeVoices” is expected to be less than or equal to “numMaxVoices”, depending on current resource usage in the system.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See Also	SL_VOICETYPE macros [see section 9.2.51]		

QueryNumberOfMIDISynthesizers			
SLresult (*QueryNumberOfMIDISynthesizers)(SLEngineCapabilitiesItf self, SLuint16 *pNumMIDISynthesizers);			
Description	Queries the number of MIDI synthesizers supported by the implementation.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pNumMIDISynthesizers	[out]	Number of MIDI synthesizers supported. Must be at least 1.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	OpenSL ES mandates support for at least one synthesizer. It does not prevent implementations from supporting more than one.		
See Also	None.		

QueryAPIVersion

```
SLresult (*QueryAPIVersion) (
    SLEngineCapabilitiesItf self,
    SLuint16 *pMajor,
    SLuint16 *pMinor,
    SLuint16 *pStep
);
```

Description	Queries the version of the OpenSL ES API implementation.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pMajor`	[out]	Major version number.
	`pMinor`	[out]	Minor version number.
	`pStep`	[out]	Step within the minor version number.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	For 1.0.1 implementations of OpenSL ES, this method should return 1, 0, and 1 for the `pMajor`, `pMinor`, and `pStep` fields, respectively.		

QueryLEDCapabilities

```
SLresult (*QueryLEDCapabilities) (
    SLEngineCapabilitiesItf self,
    SLuint32 *pIndex,
    SLuint32 *pLEDDeviceID,
    SLLEDDescriptor *pDescriptor
);
```

Description	Queries the LED array device for its capabilities.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pIndex	[in/out]	As an input, specifies which LED array device to obtain the capabilities of, the supported range is [0, *n*), where *n* is the number of LED array devices available (ignored if pDescriptor is NULL). As an output, specifies the number of LED array devices available in the system. Returns 0 if no LED array devices are available.
	pLEDDeviceId	[in/out]	If pIndex is non-NULL then returns the LED array device ID corresponding to LED array device pIndex. If pIndex is NULL then, as an input, specifies which LED array device to obtain the capabilities of (SL_DEFAULTDEVICEID_LED can be used to determine the default LED array's capabilities).
	pDescriptor	[out]	Structure defining the capabilities of the LED array device.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		

QueryLEDCapabilities	
Comments	An application can determine the number of LED array devices by calling this method with `pDescriptor` set to NULL and examining `pIndex`. The application can then determine the capabilties of all the LED array devices by calling this method multiple times with `pIndex` pointing to each different index from 0 up to one less than the number of LED array devices. An LED array device is selected using the `CreateLEDDevice()` method.
See also	`SL_DEFAULTDEVICEID_LED` [see section 9.2.13])

QueryVibraCapabilities

```
SLresult (*QueryVibraCapabilities) (
   SLEngineCapabilitiesItf self,
   SLuint32 *pIndex,
   SLuint32 *pVibraDeviceID,
   SLVibraDescriptor *pDescriptor
);
```

Description	Queries the vibration device for its capabilities.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pIndex	[in/out]	As an input, specifies which vibration device to obtain the capabilities of, the supported range is [0, *n*), where *n* is the number of vibration y devices available (ignored if pDescriptor is NULL). As an output, specifies the number of vibration devices available in the system. Returns 0 if no vibration devices are available.
	pVibraDeviceId	[in/out]	If pIndex is non-NULL then returns the vibration device ID corresponding to vibration device pIndex. If pIndex is NULL then, as an input, specifies which vibration device to obtain the capabilities of (SL_DEFAULTDEVICEID_VIBRA can be used to determine the default vibration device's capabilities).
	pDescriptor	[out]	Structure defining the capabilities of the vibration device.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	An application can determine the number of vibration devices by calling this method with pDescriptor set to NULL and examining pIndex. The application can then determine the capabilties of all the vibration devices by calling this method multiple times with pIndex pointing to each different indexes from 0 up to one less than the number of vibration devices. A vibration device is selected using the CreateVibraDevice() method.		
See also	SL_DEFAULTDEVICEID_VIBRA [see section 9.2.13])		

IsThreadSafe			
`SLresult (*IsThreadSafe) ( SLEngineCapabilitiesItf self, SLboolean *pIsThreadSafe );`			
Description	Gets the thread-safety status of the OpenSL ES implementation.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pIsThreadSafe	[out]	SL_BOOLEAN_TRUE if the implementation is thread-safe; otherwise SL_BOOLEAN_FALSE.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		

8.23 SLEnvironmentalReverbItf

Description

A sound generated within a particular acoustic environment typically reaches a listener via many different paths. The listener first hears the direct sound from the sound source itself. Somewhat later, he or she hears a number of discrete echoes caused by sound bouncing once off nearby walls, the ceiling or the floor. These sounds are known as *early reflections*. Later still, as sound waves arrive after undergoing more and more reflections, the individual reflections become indistinguishable from one another and the listener hears continuous *reverberation* that decays over time. This combination of direct, reflected and reverberant sound is illustrated in the diagram below. Please note that the reflections level and reverb level are total (integrated) energy levels of early reflections and late reverberation, respectively, not peak levels.

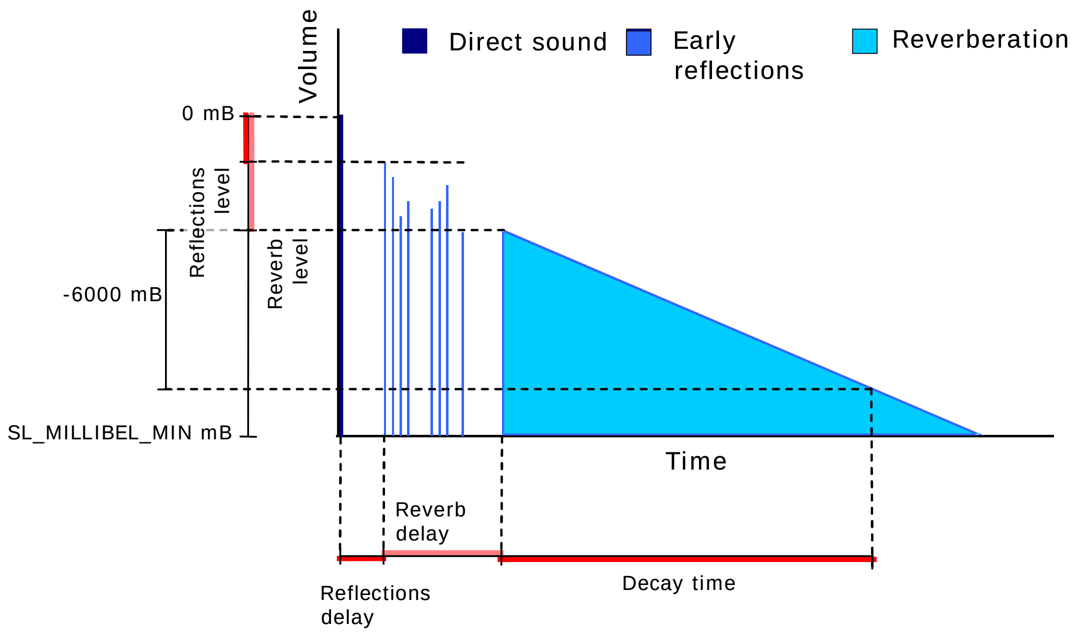


Figure 35: Reflections and reverberation

This interface allows an application to control these properties in a global reverb environment. The reverb controls exposed by this interface are based on the I3DL2 guidelines [I3DL2], with the restriction that the high-frequency reference level is fixed at 5 kHz.

When this interface is exposed on the Output Mix, it acts as an auxiliary effect; for reverb to be applied to a player's output, the `SLEffectSendItf` interface [see section 8.17] must be exposed on the player.

The following restriction must be adhered to when exposing this interface:

- It is not possible to expose this interface while the `SLPresetReverbItf` interface of the same object is already exposed.

This interface is supported on the Output Mix [see section 7.9] object.

See sections B.6.1 and C.4 for examples using this interface.

High Frequency Attenuation

When a sound is reflected against objects within a room, its high frequencies typically tail off faster than its low frequencies. This interface exposes control over how much attenuation is applied to the high frequency components of the reverb and reflections. The attenuation is controlled using a setting that specifies the attenuation level (in millibels) of the frequencies above 5 kHz (high frequencies) relative to low frequencies. This allows different implementations to use different filter designs (such as one-pole, two-pole) for the internal low-pass filter.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_ENVIRONMENTALREVERB;

struct SLEnvironmentalReverbItf_;
typedef const struct SLEnvironmentalReverbItf_ * const *
SLEnvironmentalReverbItf;

struct SLEnvironmentalReverbItf_ {
   SLresult (*SetRoomLevel) (
         SLEnvironmentalReverbItf self,
         SLmillibel room
   );
   SLresult (*GetRoomLevel) (
         SLEnvironmentalReverbItf self,
         SLmillibel *pRoom
   );
   SLresult (*SetRoomHFLevel) (
         SLEnvironmentalReverbItf self,
         SLmillibel roomHF
   );
   SLresult (*GetRoomHFLevel) (
         SLEnvironmentalReverbItf self,
         SLmillibel *pRoomHF
   );
   SLresult (*SetDecayTime) (
         SLEnvironmentalReverbItf self,
         SLmillisecond decayTime
   );
   SLresult (*GetDecayTime) (
         SLEnvironmentalReverbItf self,
         SLmillisecond *pDecayTime
   );
```

```
    SLresult (*SetDecayHFRatio) (
          SLEnvironmentalReverbItf self,
          SLpermille decayHFRatio
    );
    SLresult (*GetDecayHFRatio) (
          SLEnvironmentalReverbItf self,
          SLpermille *pDecayHFRatio
    );
    SLresult (*SetReflectionsLevel) (
          SLEnvironmentalReverbItf self,
          SLmillibel reflectionsLevel
    );
    SLresult (*GetReflectionsLevel) (
          SLEnvironmentalReverbItf self,
          SLmillibel *pReflectionsLevel
    );
    SLresult (*SetReflectionsDelay) (
          SLEnvironmentalReverbItf self,
          SLmillisecond reflectionsDelay
    );
    SLresult (*GetReflectionsDelay) (
          SLEnvironmentalReverbItf self,
          SLmillisecond *pReflectionsDelay
    );
    SLresult (*SetReverbLevel) (
          SLEnvironmentalReverbItf self,
          SLmillibel reverbLevel
    );
    SLresult (*GetReverbLevel) (
          SLEnvironmentalReverbItf self,
          SLmillibel *pReverbLevel
    );
    SLresult (*SetReverbDelay) (
          SLEnvironmentalReverbItf self,
          SLmillisecond reverbDelay
    );
    SLresult (*GetReverbDelay) (
          SLEnvironmentalReverbItf self,
          SLmillisecond *pReverbDelay
    );
    SLresult (*SetDiffusion) (
          SLEnvironmentalReverbItf self,
          SLpermille diffusion
    );
    SLresult (*GetDiffusion) (
          SLEnvironmentalReverbItf self,
          SLpermille *pDiffusion
    );
    SLresult (*SetDensity) (
          SLEnvironmentalReverbItf self,
          SLpermille density
    );
    SLresult (*GetDensity) (
          SLEnvironmentalReverbItf self,
          SLpermille *pDensity
    );
    SLresult (*SetEnvironmentalReverbProperties) (
          SLEnvironmentalReverbItf self,
          const SLEnvironmentalReverbSettings *pProperties
    );
    SLresult (*GetEnvironmentalReverbProperties) (
          SLEnvironmentalReverbItf self,
          SLEnvironmentalReverbSettings *pProperties
    );
 };
```

Interface ID

c2e5d5f0-94bd-4763-9cac-4e23-4d06839e

Defaults

Room level: SL_MILLIBEL_MIN mB

Room HF level: 0 mB

Decay time: 1000 ms

Decay HF ratio: 500 ‰

Reflections delay: 20 ms

Reflections level: SL_MILLIBEL_MIN mB

Reverb level: SL_MILLIBEL_MIN mB

Reverb delay: 40 ms

Diffusion: 1000 ‰

Density: 1000 ‰

Methods

SetRoomLevel

```
SLresult (*SetRoomLevel)(
    SLEnvironmentalReverbItf self,
    SLmillibel room
);
```

Description	Sets the master volume level of the environmental reverb effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	room	[in]	Room level in millibels. The valid range is [SL_MILLIBEL_MIN, 0].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	None.		

GetRoomLevel

```
SLresult (*GetRoomLevel) (
    SLEnvironmentalReverbItf self,
    SLmillibel *pRoom
);
```

Description	Gets the master volume level of the environmental reverb effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pRoom	[out]	Pointer to a location for the room level in millibels. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetRoomHFLevel

```
SLresult (*SetRoomHFLevel) (
    SLEnvironmentalReverbItf self,
    SLmillibel roomHF
);
```

Description	Sets the volume level at 5 kHz relative to the volume level at low frequencies of the overall reverb effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	roomHF	[in]	High frequency attenuation level in millibels. The valid range is [SL_MILLIBEL_MIN, 0].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	This controls a low-pass filter that will reduce the level of the high-frequency.		

GetRoomHFLevel

```
SLresult (*GetRoomHFLevel) (
    SLEnvironmentalReverbItf self,
    SLmillibel *pRoomHF
);
```

Description	Gets the room HF level.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pRoomHF	[out]	Pointer to a location for the room HF level in millibels. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetDecayTime

```
SLresult (*SetDecayTime) (
    SLEnvironmentalReverbItf self,
    SLmillisecond decayTime
);
```

Description	Sets the time taken for the level of reverberation to decay by 60 dB.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`decayTime`	[in]	Decay time in milliseconds. The valid range is [100, 20000].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	None.		

GetDecayTime

```
SLresult (*GetDecayTime) (
    SLEnvironmentalReverbItf self,
    SLmillisecond *pDecayTime
);
```

Description	Gets the decay time.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pDecayTime`	[out]	Pointer to a location for the decay time in milliseconds. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetDecayHFRatio

```
SLresult (*SetDecayHFRatio) (
    SLEnvironmentalReverbItf self,
    SLpermille decayHFRatio
);
```

Description	Sets the ratio of high frequency decay time (at 5 kHz) relative to the decay time at low frequencies.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`decayHFRatio`	[in]	High frequency decay ratio using a permille scale. The valid range is [100, 2000]. A ratio of 1000 indicates that all frequencies decay at the same rate.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_CONTROL_LOST`		

Comments	None.		

GetDecayHFRatio

```
SLresult (*GetDecayHFRatio) (
    SLEnvironmentalReverbItf self,
    SLpermille *pDecayHFRatio
);
```

Description	Gets the ratio of high frequency decay time (at 5 kHz) relative to low frequencies.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pDecayHFRatio`	[out]	Pointer to receive the decay HF ratio. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

SetReflectionsLevel

```
SLresult (*SetReflectionsLevel) (
    SLEnvironmentalReverbItf self,
    SLmillibel reflectionsLevel
);
```

Description	Sets the volume level of the early reflections.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`reflectionsLevel`	[in]	Reflection level in millibels. The valid range is [SL_MILLIBEL_MIN, 1000].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	This level is combined with the overall room level (set using `SetRoomLevel`).		

GetReflectionsLevel

```
SLresult (*GetReflectionsLevel) (
    SLEnvironmentalReverbItf self,
    SLmillibel *pReflectionsLevel
);
```

Description	Gets the reflections level.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pReflectionsLevel`	[out]	Pointer to the reflection level in millibels. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetReflectionsDelay

```
SLresult (*SetReflectionsDelay) (
    SLEnvironmentalReverbItf self,
    SLmillisecond reflectionsDelay
);
```

Description	Sets the delay time for the early reflections.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	reflectionsDelay	[in]	Reflections delay in milliseconds. The valid range is [0, 300].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	This method sets the time between when the direct path is heard and when the first reflection is heard.		

GetReflectionsDelay

```
SLresult (*GetReflectionsDelay) (
    SLEnvironmentalReverbItf self,
    SLmillisecond *pReflectionsDelay
);
```

Description	Gets the reflections delay.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pReflectionsDelay	[out]	Pointer to a location to receive reflections delay in milliseconds. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetReverbLevel

```
SLresult (*SetReverbLevel) (
    SLEnvironmentalReverbItf self,
    SLmillibel reverbLevel
);
```

Description	Sets the volume level of the late reverberation.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`reverbLevel`	[in]	Reverb level in millibels. The valid range is [`SL_MILLIBEL_MIN`, 2000].

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_CONTROL_LOST`		

Comments	This level is combined with the overall room level (set using `SetRoomLevel`).		

GetReverbLevel

```
SLresult (*GetReverbLevel) (
    SLEnvironmentalReverbItf self,
    SLmillibel *pReverbLevel
);
```

Description	Gets the reverb level.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pReverbLevel`	[out]	Pointer to a location for the reverb level in millibels. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

SetReverbDelay

```
SLresult (*SetReverbDelay) (
    SLEnvironmentalReverbItf self,
    SLmillisecond reverbDelay
);
```

Description	Sets the time between the first reflection and the reverberation.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`reverbDelay`	[in]	Reverb delay in milliseconds. The valid range is [0, 100].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	None.		

GetReverbDelay

```
SLresult (*GetReverbDelay) (
    SLEnvironmentalReverbItf self,
    SLmillisecond *pReverbDelay
);
```

Description	Gets the reverb delay length.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pReverbDelay`	[out]	Pointer to the location for the reverb delay in milliseconds. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetDiffusion

```
SLresult (*SetDiffusion) (
    SLEnvironmentalReverbItf self,
    SLpermille diffusion
);
```

Description	Sets the echo density in the late reverberation decay.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`diffusion`	[in]	Diffusion specified using a permille scale. The valid range is [0, 1000]. A value of 1000 ‰ indicates a smooth reverberation decay. Values below this level give a more *grainy* character.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	The scale should approximately map linearly to the perceived change in reverberation.		

GetDiffusion

```
SLresult (*GetDiffusion) (
    SLEnvironmentalReverbItf self,
    SLpermille *pDiffusion
);
```

Description	Gets the level of diffusion.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pDiffusion`	[out]	Pointer to a location for the diffusion level. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetDensity

```
SLresult (*SetDensity) (
    SLEnvironmentalReverbItf self,
    SLpermille density
);
```

Description	Controls the modal density of the late reverberation decay.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	density	[in]	Density specified using a permille scale. The valid range is [0, 1000]. A value of 1000 ‰ indicates a natural sounding reverberation. Values below this level produce a more colored effect.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	The scale should approximately map linearly to the perceived change in reverberation. A lower density creates a hollow sound that is useful for simulating small reverberation spaces such as bathrooms.		

GetDensity

```
SLresult (*GetDensity) (
    SLEnvironmentalReverbItf self,
    SLpermille *pDensity
);
```

Description	Gets the density level.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pDensity	[out]	Pointer to a location for the density level. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetEnvironmentalReverbProperties			
`SLresult (*SetEnvironmentalReverbProperties) ( SLEnvironmentalReverbItf self, const SLEnvironmentalReverbSettings *pProperties );`			
Description	Sets all the environment properties in one method call.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pProperties	[in]	Pointer to a structure containing all the environmental reverb properties [see section 9.1.21]. All the properties in the structure must be within the ranges specified for each of the properties in the "set" methods in this interface. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		

Comments	This can be used with the environmental reverb presets definitions. For example:
	`SLEnvironmentalReverbSettings ReverbSettings = SL_I3DL2_ENVIRONMENT_PRESET_BATHROOM;`
	`/* Change reverb environment to bathroom. */`
	`pReverb->SetEnvironmentalReverbProperties(pReverb, &ReverbSettings);`
	Developers are advised to use this method when changing more than one parameter at a given time as this reduces the amount of unnecessary processing.		

```
SLresult (*SetEnvironmentalReverbProperties) (
    SLEnvironmentalReverbItf self,
    const SLEnvironmentalReverbSettings *pProperties
);
```

```
SLEnvironmentalReverbSettings ReverbSettings =
    SL_I3DL2_ENVIRONMENT_PRESET_BATHROOM;

/* Change reverb environment to bathroom. */
pReverb->SetEnvironmentalReverbProperties(pReverb,
&ReverbSettings);
```

GetEnvironmentalReverbProperties			
`SLresult (*GetEnvironmentalReverbProperties) ( SLEnvironmentalReverbItf self, SLEnvironmentalReverbSettings *pProperties );`			
Description	Gets all the environment's properties.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pProperties	[in]	Pointer to a structure to receive all the environmental reverb properties [see section 9.1.21]. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.24 SLEqualizerItf

Description

SLEqualizerItf is an interface for manipulating the equalization settings of a media object. The equalizer (EQ) can be set up in two different ways: by setting individual frequency bands, or by using predefined presets.

The preset settings can be directly taken into use with the method UsePreset(). The current preset can be queried with the method GetPreset(). If none of the presets is set, SL_EQUALIZER_UNDEFINED will be returned. SL_EQUALIZER_UNDEFINED will also be returned when a preset has been set, but the equalizer settings have been altered later with SetBandLevel(). Presets have names that can be used in the user interface.

There are methods for getting and setting individual EQ-band gains (SetBandLevel() and GetBandLevel()), methods for querying the number of the EQ-bands available (GetNumberOfBands()) and methods for querying their center frequencies (GetCenterFreq()).

The gains in this interface are defined in millibels (hundredths of a decibel), but it has to be understood that many devices contain a Dynamic Range Control (DRC) system that will affect the actual effect and therefore the value in millibels will affect as a guideline rather than as a strict rule.

This interface affects different parts of the audio processing chain, depending on which object the interface is exposed. If this interface is exposed on an Output Mix object, the effect is applied to the output mix. If this interface is exposed on a Player object, it is applied to the Player's output only. For more information, see section 4.5.1.

This interface is supported on the Output Mix [see section 7.9] object.

See section B.6.2 for an example using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_EQUALIZER;

struct SLEqualizerItf_;
typedef const struct SLEqualizerItf_ * const * SLEqualizerItf;

struct SLEqualizerItf_ {
   SLresult (*SetEnabled)(
         SLEqualizerItf self,
         SLboolean enabled
   );
```

```
      SLresult (*IsEnabled)(
            SLEqualizerItf self,
            SLboolean *pEnabled
      );
      SLresult (*GetNumberOfBands)(
            SLEqualizerItf self,
            SLuint16 *pNumBands
      );
      SLresult (*GetBandLevelRange)(
            SLEqualizerItf self,
            SLmillibel *pMin,
            SLmillibel *pMax
      );
      SLresult (*SetBandLevel)(
            SLEqualizerItf self,
            SLuint16 band,
            SLmillibel level
      );
      SLresult (*GetBandLevel)(
            SLEqualizerItf self,
            SLuint16 band,
            SLmillibel *pLevel
      );
      SLresult (*GetCenterFreq)(
            SLEqualizerItf self,
            SLuint16 band,
            SLmilliHertz *pCenter
      );
      SLresult (*GetBandFreqRange)(
            SLEqualizerItf self,
            SLuint16 band,
            SLmilliHertz *pMin,
            SLmilliHertz *pMax
      );
      SLresult (*GetBand)(
            SLEqualizerItf self,
            SLmilliHertz frequency,
            SLuint16 *pBand
      );
      SLresult (*GetCurrentPreset)(
            SLEqualizerItf self,
            SLuint16 *pPreset
      );
      SLresult (*UsePreset)(
            SLEqualizerItf self,
            SLuint16 index
      );
      SLresult (*GetNumberOfPresets)(
            SLEqualizerItf self,
            SLuint16 *pNumPresets
      );
      SLresult (*GetPresetName)(
            SLEqualizerItf self,
            SLuint16 index,
            SLuint16* pSize,
            SLchar * pName
      );
   };
```

Interface ID

09303cf0-d033-11df-bd3b-0800200c9a66

Defaults

Enabled: false (disabled)

All band levels: 0 mB (flat response curve)

Preset: SL_EQUALIZER_UNDEFINED (no preset)

Methods

SetEnabled

```
   SLresult (*SetEnabled)(
      SLEqualizerItf self,
      SLboolean enabled
   );
```

Description	Enables the effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	enabled	[in]	True to turn on the effect; false to switch it off.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_CONTROL_LOST		
Comments	None.		

IsEnabled

```
SLresult (*IsEnabled)(
    SLEqualizerItf self,
    SLboolean *pEnabled
);
```

Description	Gets the enabled status of the effect.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pEnabled`	[out]	True if the effect is on, otherwise false. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

GetNumberOfBands

```
SLresult (*GetNumberOfBands)(
    SLEqualizerItf self,
    SLuint16 *pNumBands
);
```

Description	Gets the number of frequency bands that the equalizer supports. A valid equalizer must have at least two bands.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pNumBands`	[out]	Number of frequency bands that the equalizer supports. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

GetBandLevelRange

```
SLresult (*GetBandLevelRange)(
   SLEqualizerItf self,
   SLmillibel *pMin,
   SLmillibel *pMax
);
```

Description	Returns the minimum and maximum band levels supported.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMin	[out]	Minimum supported band level in millibels.
	pMax	[out]	Maximum supported band level in millibels.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The range returned by GetBandLevelRange must at least include 0 mB. The application may pass NULL as one of the [out] parameters to find out only the other one's value.		

SetBandLevel			

`SLresult (*SetBandLevel)(`
`    SLEqualizerItf self,`
`    SLuint16 band,`
`    SLmillibel level`
`);`			

Description	Sets the given equalizer band to the given gain value.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	band	[in]	Frequency band that will have the new gain. The numbering of the bands starts from 0 and ends at (number of bands – 1).
	level	[in]	New gain in millibels that will be set to the given band. GetBandLevelRange() will define the maximum and minimum values.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	None.		

GetBandLevel

```
SLresult (*GetBandLevel)(
   SLEqualizerItf self,
   SLuint16 band,
   SLmillibel *pLevel
);
```

Description	Gets the gain set for the given equalizer band.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`band`	[in]	Frequency band whose gain is requested. The numbering of the bands starts from 0 and ends at (number of bands – 1).
	`pLevel`	[out]	Gain in millibels of the given band. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

GetCenterFreq

```
SLresult (*GetCenterFreq)(
   SLEqualizerItf self,
   SLuint16 band,
   SLmilliHertz *pCenter
);
```

Description	Gets the center frequency of the given band.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`band`	[in]	Frequency band whose center frequency is requested. The numbering of the bands starts from 0 and ends at (number of bands – 1).
	`pCenter`	[out]	The center frequency in milliHertz. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

GetBandFreqRange

```
SLresult (*GetBandFreqRange)(
    SLEqualizerItf self,
    SLuint16 band,
    SLmilliHertz *pMin,
    SLmilliHertz *pMax
);
```

Description	Gets the frequency range of the given frequency band.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	band	[in]	Frequency band whose frequency range is requested. The numbering of the band that can be used with this method starts from 0 and ends at (number of bands – 1).
	pMin	[out]	The minimum frequency in milliHertz.
	pMax	[out]	The maximum frequency in milliHertz.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The exposed band ranges do not overlap (physically they many times do, but the virtual numbers returned here do not) - this is in order to simplify the applications that want to use this information for graphical representation of the EQ. If shelving filters are used in the lowest and the highest band of the equalizer, the lowest band returns 0 mHz as the minimum frequency and the highest band returns the SL_MILLIHERTZ_MAX as the maximum frequency. The application may pass NULL as one of the [out] parameters to find out only the other one's value.		

GetBand

```
SLresult (*GetBand)(
   SLEqualizerItf self,
   SLmilliHertz frequency,
   SLuint16 *pBand
);
```

Description	Gets the band that has the most effect on the given frequency. If no band has an effect on the given frequency, SL_EQUALIZER_UNDEFINED is returned.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	frequency	[in]	Frequency in milliHertz which is to be equalized via the returned band
	pBand	[out]	Frequency band that has most effect on the given frequency or SL_EQUALIZER_UNDEFINED if no band has an effect on the given frequency. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

GetCurrentPreset

```
SLresult (*GetCurrentPreset)(
   SLEqualizerItf self,
   SLuint16 *pPreset
);
```

Description	Gets the current preset.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pPreset	[out]	Preset that is set at the moment. If none of the presets are set, SL_EQUALIZER_UNDEFINED will be returned. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

UsePreset

```
SLresult (*UsePreset)(
    SLEqualizerItf self,
    SLuint16 index
);
```

Description	Sets the equalizer according to the given preset.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	New preset that will be taken into use. The valid range is [0, number of presets-1].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	None.		

GetNumberOfPresets

```
SLresult (*GetNumberOfPresets)(
    SLEqualizerItf self,
    SLuint16 *pNumPresets
);
```

Description	Gets the total number of presets the equalizer supports. The presets will have indices [0, number of presets-1].		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pNumPresets	[out]	Number of presets the equalizer supports. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

GetPresetName

```
SLresult (*GetPresetName)(
   SLEqualizerItf self,
   SLuint16 index,
   SLuint16 * pSize,
   SLchar * pName
);
```

Description	Gets the preset name based on the index.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	Index of the preset. The valid range is [0, number of presets-1].
	pSize	[in/out]	pSize will be returned with the array size required to store the name. If pName is not NULL, pSize must specify the maximum character count that pName is able to store. This parameter must not be NULL.
	pName	[out]	A non-empty, null terminated string containing the name of the given preset. The character coding is UTF-8.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.25 SLLEDArrayItf

Description

`SLLEDArrayItf` is used to activate / deactivate the LEDs, as well as to set the color of LEDs, if supported.

`SLLEDArrayItf` uses the following state model per LED, which indicates whether the LED is on or off:

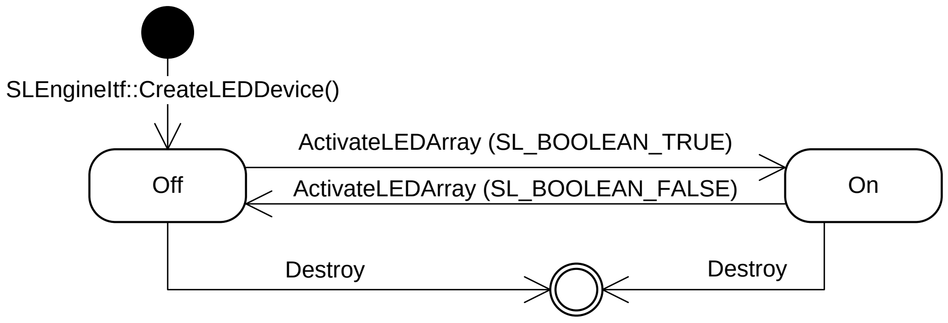

SLEngineItf::CreateLEDDevice()
ActivateLEDArray (SL_BOOLEAN_TRUE)
ActivateLEDArray (SL_BOOLEAN_FALSE)
Off
On
Destroy
Destroy

Figure 36: SLLEDArrayItf state model

This interface is supported on the LED Array [see section 8.25] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_LED;

struct SLLEDArrayItf_;
typedef const struct SLLEDArrayItf_ * const * SLLEDArrayItf;

struct SLLEDArrayItf_ {
   SLresult (*ActivateLEDArray) (
         SLLEDArrayItf self,
         SLuint32 lightMask
   );
   SLresult (*IsLEDArrayActivated) (
         SLLEDArrayItf self,
         SLuint32 *lightMask
   );
   SLresult (*SetColor) (
         SLLEDArrayItf self,
         SLuint8 index,
         const SLHSL *color
   );
   SLresult (*GetColor) (
         SLLEDArrayItf self,
         SLuint8 index,
         SLHSL *color
   );
};
```

Interface ID

2cc1cd80-ddd6-11db-807e-0002a5d5c51b

Defaults

Initially, all LEDs are in the off state. Default color is undefined.

Methods

ActivateLEDArray

```
SLresult (*ActivateLEDArray) (
    SLLEDArrayItf self,
    SLuint32 lightMask
);
```

Description	Activates or deactivates individual LEDs in an array of LEDs.		
Pre-conditions	None.		
Parameters	`self`	[in]	Pointer to a `SLLEDArrayItf` interface.
	`lightMask`	[in]	Bit mask indicating which LEDs should be activated or deactivated.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	Valid bits in `lightMask` range from the least significant bit, which indicates the first LED in the array, to bit `SLLEDDescriptor::ledCount` – 1, which indicates the last LED in the array. Bits set outside this range are ignored.		
See also	`SLLEDDescriptor` [see section 9.1.24].		

IsLEDArrayActivated

```
SLresult (*IsLEDArrayActivated) (
    SLLEDArrayItf self,
    SLuint32 *lightMask
);
```

Description	Returns the state of each LED in an array of LEDs.		
Pre-conditions	None.		
Parameters	`self`	[in]	Pointer to a `SLLEDArrayItf` interface.
	`lightMask`	[out]	Address to store a bit mask indicating which LEDs are activated or deactivated.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Valid bits in `lightMask` range from the least significant bit, which indicates the first LED in the array, to bit `SLLEDDescriptor::ledCount` – 1, which indicates the last LED in the array. Bits set outside this range are ignored.		
See also	`SLLEDDescriptor` [see section 9.1.24].		

SetColor

```
SLresult (*SetColor) (
    SLLEDArrayItf self,
    SLuint8 index,
    const SLHSL *color
);
```

Description	Sets the color of an individual LED.		
Pre-conditions	The LED must support setting color, per `SLLEDDescriptor::colorMask`.		
Parameters	`self`	[in]	Pointer to a `SLLEDArrayItf` interface.
	`index`	[in]	Index of the LED. Range is [0, `SLLEDDescriptor::ledCount`)
	`color`	[in]	Address of a data structure containing an HSL color.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_CONTROL_LOST`		

Comments	None.		
See also	`SLLEDDescriptor` [see section 9.1.24].		

GetColor

```
SLresult (*GetColor) (
   SLLEDArrayItf self,
   SLuint8 index,
   SLHSL *color
);
```

Description	Returns the color of an individual LED.		
Pre-conditions	The LED must support setting color, per `SLLEDDescriptor::colorMask`.		
Parameters	`self`	[in]	Pointer to a `SLLEDArrayItf interface`.
	`index`	[in]	Index of the LED. Range is [0, `SLLEDDescriptor::ledCount`)
	`color`	[out]	Address to store a data structure containing an HSL color.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PRECONDITIONS_VIOLATED SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	`SLLEDDescriptor` [see section 9.1.24].		

8.26 SLMetadataExtractionItf

Description

The `SLMetadataExtractionItf` interface allows an application developer to acquire metadata. It is used to scan through a file's metadata, provide the ability to determine how many metadata items are available, filter for or against metadata items by key, and to have the engine fill in a data structure containing full metadata information for a metadata item.

The `SLMetadataExtractionItf` interface defaults to a simple search: in the case of simple formats (e.g. MP3, ADTS, WAVE, AU, AIFF), there is only one location for metadata, and this simple method searches it completely; in the case of advanced formats (e.g. MP4/3GP, XMF, SMIL), there are potentially many locations for metadata, and the engine searches only the topmost layer of metadata. Used in combination with the `SLMetadataTraversalItf` interface, the `SLMetadataExtractionItf` interface is able to search all metadata in any file using a variety of search modes.

This interface is supported on the Audio Player [see section 7.2], MIDI Player [see section 7.8] and Metadata Extractor [see section 7.7] objects.

See section B.4 for an example on using this interface.

Dynamic Interface Addition

If this interface is added dynamically (using `DynamicInterfaceManagementItf`) the set of exposed metadata items might be limited compared to the set of exposed items that would exist if this interface been requested during object creation time. Typically, this might be the case in some implementations for efficiency reasons, or when the interface is added dynamically during playback of non-seekable streamed content and the metadata is located earlier in the stream than the interface addition time.

Khronos Keys

The keys that can be used to access metadata are the keys defined in the metadata specification of the media format in question. In addition, the OpenSL ES specification defines a few format-agnostic keys, called “Khronos keys”. The Khronos keys are for those developers who may not be familiar with the original metadata keys of the various media formats, but still want to extract metadata using OpenSL ES. It is the responsibility of API implementations to map these Khronos keys to the format-specific standard metadata keys. The Khronos keys are not meant to replace the standard metadata keys or to restrict the number of metadata keys available to the application. Developers conversant with the standard metadata keys in each format can still specify the exact keys in which they are interested in by using `MetadataExtractionItf`. The support for these Khronos keys is format-dependent.

The following table lists the Khronos keys. This list does not purport to be a comprehensive union of the standard keys in the various media formats. On the contrary, it is deliberately limited to the set of commonly-used metadata items. It should be considered a baseline list.

Table 10: Khronos Keys

"KhronosTitle"	The title of the low-level entity, such as the name of the song, book chapter, image, video clip.
"KhronosAlbum"	The title of the high-level entity, such as the name of the song/video/image album, the name of the book.
"KhronosTrackNumber"	The number of the track.
"KhronosArtist"	The name of the artist, performer.
"KhronosGenre"	The genre of the media.
"KhronosYear"	The release year.
"KhronosComment"	Other comments on the media. For example, for images, this could be the event at which the photo was taken.
"KhronosArtistURL"	A URL pointing to the artist's site.
"KhronosContentURL"	A URL pointing to the site from which (alternate versions of) the content can be downloaded.
"KhronosRating"	A subjective rating of the media.
"KhronosAlbumArtJPEG"	Associated JPEG image, such as album art. The value associated with this key (the image itself) is in binary, in one of several image formats.
"KhronosAlbumArtPNG"	Associated PNG image, such as album art. The value associated with this key (the image itself) is in binary, in one of several image formats.
"KhronosCopyright"	Copyright text.
"KhronosSeekPoint"	Seek points of the media.

In this regard, three important scenarios are worth considering:

Scenario 1: Some of the Khronos keys do not have an equivalent standard metadata key in the media format under consideration: Only those Khronos keys for which there exists a mapping to the standard keys of the media are populated; the remaining Khronos keys remain empty, that is, no mapping exists and they are not exposed.

Scenario 2: The application is interested in metadata keys that are not part of the list of Khronos keys: The application has the option of ignoring the Khronos keys entirely and directly specifying exactly those standard metadata keys that it is interested in, using `MetadataExtractionItf`.

Scenario 3: The application's metadata key list of interest is a proper superset of the Khronos key list: The application has the option of ignoring the Khronos key list entirely (as in Scenario #2) or it can use the Khronos key list and supplement it by accessing the extra format-specific standard keys directly using the `MetadataExtractionItf`.

All the Khronos keys are encoded in ASCII. The encoding and the language country code of the associated values depend on the media content. However, the encoding of the values is in one of the encoded strings with an exception that the values associated with "KhronosAlbumArtJPEG" and "KhronosAlbumArtPNG" keys have the encoding `SL_CHARACTERENCODING_BINARY`.

Seek Points

`SLMetadataExtractionItf` can be used for querying the seek points of the media. This is done by using the standard metadata (ASCII) key "KhronosSeekPoint".

The associated value of Khronos seek points are represented with `SLMetadataInfo` structures, which is the case with all the metadata keys. The character encoding of this `SLMetadataInfo` structure is `SL_CHARACTERENCODING_BINARY`, since the value has special format described below.

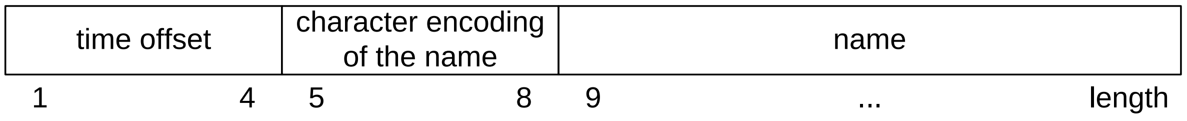


Figure 37: The data field of SLMetadataInfo Structure containing the value corresponding to a KhronosSeekPoint key.

The data field of the `SLMetadataInfo` struct contains in its first 4 bytes the time offset (little endian) of the seek point as `SLmilliseconds`. (The length of the value is 4 bytes, since `SLmillisecond` is `SLuint32`.) `SeekItf::SetPosition()` can be used for seeking that seek point.

The bytes from the 5th to the 8th contain the character encoding of the name of the seek point as a `SL_CHARACTERENCODING` macro.

Starting from the 9th byte, the data field contains the name of the seek point (for example, the chapter name) in the character encoding defined in bytes 5 to 8 and the language defined in the `SLMetadataInfo` struct. The name is always null-terminated, which means that even if the name would be empty, the length of the value is always at least 9 bytes.

There can be multiple "KhronosSeekPoint" items for the same seek point to allow multiple language support. That is, the number of "KhronosSeekPoint" items is the number of seek points times the number of languages supported. The `AddKeyFilter()` method can be used for looking at seek points only in specific language by setting the pKey parameter as

"KhronosSeekPoint" and the `valueLangCountry` parameter to contain the language / country code of interest.

Mandated Keys

An implementation of `SLMetadataExtractionItf` must support all methods on the interface. This specification does not mandate that an implementation support any particular key (Khronos key or otherwise) even in cases where the interface itself is mandated on an object.

Filtering of Metadata Items

The interface enables filtering of metadata items according to several criteria (see `AddKeyFilter()`). Theoretically, the application may never use the filtering functionality and do filtering itself. However, in practice, an implementation may use the filtering information in order to make extraction more efficient in terms of memory consumption or computational complexity. It is recommended that applications that are not interested in the entire set of metadata items will use the filtering mechanism.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_METADATAEXTRACTION;

struct SLMetadataExtractionItf_;
typedef const struct SLMetadataExtractionItf_
         * const * SLMetadataExtractionItf;

struct SLMetadataExtractionItf_ {
   SLresult (*GetItemCount) (
         SLMetadataExtractionItf self,
         SLuint32 *pItemCount
   );
   SLresult (*GetKeySize) (
         SLMetadataExtractionItf self,
         SLuint32 index,
         SLuint32 *pKeySize
   );
   SLresult (*GetKey) (
         SLMetadataExtractionItf self,
         SLuint32 index,
         SLuint32 keySize,
         SLMetadataInfo *pKey
   );
   SLresult (*GetValueSize) (
         SLMetadataExtractionItf self,
         SLuint32 index,
         SLuint32 *pValueSize
   );
```

```
    SLresult (*GetValue) (
          SLMetadataExtractionItf self,
          SLuint32 index,
          SLuint32 valueSize,
          SLMetadataInfo *pValue
    );
    SLresult (*AddKeyFilter) (
          SLMetadataExtractionItf self,
          SLuint32 keySize,
          const void *pKey,
          SLuint32 keyEncoding,
          const SLchar *pValueLangCountry,
          SLuint32 valueEncoding,
          SLuint8 filterMask
    );
    SLresult (*ClearKeyFilter) (
          SLMetadataExtractionItf self
    );
};
```

Interface ID

aa5b1f80-ddd6-11db-ac8e-0002a5d5c51b

Defaults

The metadata key filter is empty upon realization of the interface. The default metadata scope is the root of the file.

Methods

GetItemCount			
`SLresult (*GetItemCount) ( SLMetadataExtractionItf self, SLuint32 *pItemCount );`			
Description	Returns the number of metadata items within the current scope of the object.		
Pre-conditions	None		
Parameters	self	[in]	Interface self-reference.
	pItemCount	[out]	Number of metadata items. Must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	itemCount is determined by the current metadata filter. For example, in a situation where four metadata items exist, and there is no filter, GetItemCount() will return 4; if there is a filter that matched only one of the keys, GetItemCount() will return 1. GetItemCount() returns the number of metadata items for a given metadata scope (active node). The scope is determined by methods within SLMetadataTraversalItf.		
See also	None.		

GetKeySize

```
SLresult (*GetKeySize) (
    SLMetadataExtractionItf self,
    SLuint32 index,
    SLuint32 *pKeySize
);
```

Description	Returns the byte size of a given metadata key.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	Metadata item Index. Range is [0, GetItemCount()).
	pKeySize	[out]	Address to store key size. size must be greater than 0. Must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	GetKeySize() is used for determining how large a block of memory is necessary to hold the key returned by GetKey().		
See also	GetKey()		

GetKey

```
SLresult (*GetKey) (
    SLMetadataExtractionItf self,
    SLuint32 index,
    SLuint32 keySize,
    SLMetadataInfo *pKey
);
```

Description	Returns a SLMetadataInfo structure and associated data referenced by the structure for a key.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	Metadata item Index. Range is [0, GetItemCount()).
	keySize	[in]	Size of the memory block passed as key. Range is [1, GetKeySize()].
	pKey	[out]	Address to store the key. Must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_BUFFER_INSUFFICIENT		
Comments	GetKey() fills out the SLMetadataInfo structure, including data for the key beyond the size of the structure. If the given size is smaller than the needed size SL_RESULT_BUFFER_INSUFFICIENT is returned and only data of the given size will be written; however, no invalid strings are written. That is, the null-terminator always exists and multibyte characters are not cut in the middle.		
See also	GetKeySize()		

GetValueSize

```
SLresult (*GetValueSize) (
   SLMetadataExtractionItf self,
   SLuint32 index,
   SLuint32 *pValueSize
);
```

Description	Returns the byte size of a given metadata value.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	Metadata item Index. Range is [0, GetItemCount()).
	pValueSize	[out]	Address to store value size. size must be greater than 0. Must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	GetValueSize() is used for determining how large a block of memory is necessary to hold the value returned by GetValue().		
See also	GetValue()		

GetValue

```
SLresult (*GetValue) (
   SLMetadataExtractionItf self,
   SLuint32 index,
   SLuint32 size,
   SLMetadataInfo *pValue
);
```

Description	Returns a `SLMetadataInfo` structure and associated data referenced by the structure for a value.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`index`	[in]	Metadata item Index. Range is [0, `GetItemCount()`).
	`size`	[in]	Size of the memory block passed as `value`. Range is [0, `GetValueSize()`].
	`pValue`	[out]	Address to store the `value`. Must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_BUFFER_INSUFFICIENT		
Comments	`GetValue()` fills out the `SLMetadataInfo` structure, including data for the value beyond the size of the structure. If the given size is smaller than the needed size `SL_RESULT_BUFFER_INSUFFICIENT` is returned and only data of the given size will be written; however, no invalid strings are written. That is, the null-terminator always exists and multibyte characters are not cut in the middle.		
See also	`GetValueSize()`		

AddKeyFilter

```
SLresult (*AddKeyFilter) (
   SLMetadataExtractionItf self,
   SLuint32 keySize,
   const void *pKey,
   SLuint32 keyEncoding,
   const SLchar *pValueLangCountry,
   SLuint32 valueEncoding,
   SLuint8 filterMask
);
```

Description	Adds a filter for a specific key.		
Pre-conditions	At least one criteria parameter (pKey, keyEncoding, pValueLangCountry, valueEncoding) must be provided.		
Parameters	self	[in]	Interface self-reference.
	keySize	[in]	Size, in bytes, of the pKey parameter. Ignored if filtering by key is disabled.
	pKey	[in]	The key to filter by. The entire key must match. Ignored if filtering by key is disabled.
	keyEncoding	[in]	Character encoding of the pKey parameter. Ignored if filtering by key is disabled.
	pValueLangCountry	[in]	Language / country code of the value to filter by. Ignored if filtering by language / country is disabled. See SLMetadataInfo structure in section 9.1.25.
	valueEncoding	[in]	Encoding of the value to filter by. Ignored if filtering by encoding is disabled.
	filterMask	[in]	Bitmask indicating which criteria to filter by. Should be one of the SL_METADATA_FILTER macros, see section 9.2.22.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	AddKeyFilter() adds a key to the metadata key filter. The filter defines which metadata items are available when asking how many exist (GetItemCount()) and how they are indexed for calls to GetKeySize(), GetKey(), GetValueSize(), and GetValue(). For example, if a file contains two metadata items, with keys "foo" and "bar" (both ASCII), calling AddKeyFilter() for "foo" will cause GetItem to return only the metadata item "foo." A subsequent call to AddKeyFilter for "bar" will cause GetItem() to return both metadata items.		

AddKeyFilter	
	The key filter uses one or more of the following criteria: key data, value encoding, and language country specification. Key data filter will consider a metadata item to match when the data in the filter key charset encoding and filter key value fields are identical to the key charset encoding and key value, respectively, found in a metadata item in the media. If the filter key charset encoding is different from the charset encoding that the media metadata item uses, it is optional for the implementation to convert the values of the filter key and the media metadata item key to the same charset, and evaluate whether they match. Language / country filter will consider a metadata item to match the criteria if the item's value language / country matches the filter's language / country code. Refer to above for a description of what matching means. The value encoding filter will simply match all items with the same value encoding. While it is possible to use all three criteria when calling `AddKeyFilter()`, it is also possible to include fewer criteria. `filterMask` is used for defining which criteria should be considered when calling `AddKeyFilter()`. It is constructed by bit-wise ORing of the metadata filter macros, see section 9.2.22. Note that `AddKeyFilter()` treats parameters as if they were ANDed together. For example, calling `AddKeyFilter()` with key data and language / country code (but not encoding) means that the filter will cause metadata that matches both the key and the language / country code to be returned, but not metadata that matches only one. Further note that subsequent calls to `AddKeyFilter()` are treated as if they were ORed together. For example, if the first call passed a key (but nothing else) and a second call passed a key and an encoding (but no language / country code), the interface will return metadata matching the first key and metadata matching both the second key and the encoding. For example, to filter for all metadata that uses the ASCII encoding for the value, pass `valueEncoding` as `SL_CHARACTERENCODING_ASCII` and `filterMask` as `SL_METADATA_FILTER_ENCODING`. To filter for all metadata that uses the ASCII encoding for the value *and* uses the language country code "en-us", pass `valueEncoding` as `SL_CHARACTERENCODING_ASCII`, `valueLangCountry` as "en-us," and `filterMask` as `SL_METADATA_FILTER_ENCODING` | `SL_METADATA_FILTER_LANG`. Note that when the filter is clear (that is, when no filter criteria have been added or after they have been cleared), the filter is defined so that `GetItemCount()` returns all metadata items (as if each criteria was set to a wildcard).
See also	`ClearKeyFilter()`

ClearKeyFilter			
`SLresult (*ClearKeyFilter) ( SLMetadataExtractionItf self );`			
Description	Clears the key filter.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS		
Comments	Note that when the filter is clear (that is, when no filter criteria have been added or after they have been cleared), the filter is defined so that GetItemCount() returns all metadata items (as if each criteria was set to a wildcard).		
See also	AddKeyFilter()		

8.27 SLMetadataMessageItf

Description

The Metadata message interface is used to set metadata callbacks. Via the callback the user will be notified of any metadata that is encountered during playback.

This interface on its own has no function or utility. It is designed for use in conjunction with SLMetadataExtractionItf, and is only able to be realized or dynamically added to an object when SLMetadataExtractionItf is already present.

The configuration state of the paired SLMetadataExtractionItf directly effects the performance of SLMetadataMessageItf. For example any filters that are active will cause metadata during playback to be filtered prior to its exposure via callback.

This interface is supported on the Audio Player [see section 7.2], Metadata Extractor [see section 7.7] and MIDI Player [see section 7.8].

Prototype

```
SL_API extern const SLInterfaceID SL_IID_METADATAMESSAGE;

struct SLMetadataMessageItf_;
typedef const struct SLMetadataMessageItf_ * const *
SLMetadataMessageItf;
```

```
struct SLMetadataMessageItf_ {
   SLresult (*RegisterMetadataCallback) (
         SLMetadataMessageItf self,
         slMetadataCallback callback,
         void *pContext
   );
};
```

Interface ID

069ff460-c5db-11df-bd3b-0800200c9a66

Callbacks

slMetadataCallback

```
   typedef void (SLAPIENTRY *slMetadataCallback){
         SLMetadataMessageItf caller,
         void *pContext,
         SLuint32 index
   );
```

Description	Executes when a Metadata element is encountered during playback.		
Parameters	caller	[in]	Interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	index	[in]	Metadata item Index for the metadata encountered. Interaction with this Index is done via the `SLMetadataExtractionItf` interface that must accompany the `SLMetadataMessageItf` interface. Additionally, this Index is only guaranteed to be valid and fixed for the duration of the callback.

Methods

RegisterMetadataCallback

```
SLresult (*RegisterMetadataCallback) (
      SLMetadataMessageItf self,
      slMetadataCallback callback,
      void *pContext
);
```

Description	Sets or clears the player’s metadata callback.		
Parameters	self	[in]	Interface self-reference
	callback	[in]	Address of the metadata callback.
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
See also	slMetadataCallback		

8.28 SLMetadataTraversalItf

Description

The SLMetadataTraversalItf interface is used in order to support advanced metadata extraction. It allows developers to traverse a file using a variety of modes, which determine how to traverse the metadata and define how the methods within the interface behave.

The interface provides the ability to set the traversal mode, to determine how many child nodes exist within a given scope and what their type is, and to set the scope.

This interface is supported on the Audio Player [see section 7.2], MIDI Player [see section 7.8] and Metadata Extractor [see section 7.7] objects.

Dynamic Interface Addition

If this interface is added dynamically (using SLDynamicInterfaceManagementItf [see section 8.17]) the set of exposed metadata nodes might be limited compared to the set of exposed nodes had this interface been requested during object creation time. Typically, this might be the case in some implementations for efficiency reasons, or when the interface is added dynamically during playback of non-seekable streamed content and the metadata is located earlier in the stream than what was the interface addition time.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_METADATATRAVERSAL;

struct SLMetadataTraversalItf_;
typedef const struct SLMetadataTraversalItf_
         * const * SLMetadataTraversalItf;

struct SLMetadataTraversalItf_ {
   SLresult (*SetMode) (
         SLMetadataTraversalItf self,
         SLuint32 mode
   );
   SLresult (*GetChildCount) (
         SLMetadataTraversalItf self,
         SLuint32 *pCount
   );
   SLresult (*GetChildMIMETypeSize) (
         SLMetadataTraversalItf self,
         SLuint32 index,
         SLuint32 *pSize
   );
   SLresult (*GetChildInfo) (
         SLMetadataTraversalItf self,
         SLuint32 index,
         SLint32 *pNodeID,
         SLuint32 *pType,
         SLuint32 size,
         SLchar *pMimeType
   );
   SLresult (*SetActiveNode) (
         SLMetadataTraversalItf self,
         SLuint32 index
   );
};
```

Interface ID

c43662c0-ddd6-11db-a7ab-0002a5d5c51b

Defaults

The metadata traversal mode defaults to SL_METADATATRAVERSALMODE_NODE [see section 9.2.23]. The default metadata scope is the root of the file. That is, the active node is root by default.

Methods

SetMode			
`SLresult (*SetMode) ( SLMetadataTraversalItf self, SLuint32 mode );`			
Description	Sets the metadata traversal mode.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	mode	[in]	Mode of metadata traversal. Must be one of the SL_METADATATRAVERSALMODE macros.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Metadata traversal mode determines how a file is parsed for metadata. It is possible to traverse the file either by iterating through the file in tree fashion - by node (SL_METADATATRAVERSALMODE_NODE, the default mode), or by scanning through the file as if it were a flat list of metadata items (SL_METADATATRAVERSALMODE_ALL). The optimal mode is largely determined by the file format.		
See also	SL_METADATATRAVERSALMODE [see section 9.2.23]		

GetChildCount			
`SLresult (*GetChildCount) ( SLMetadataTraversalItf self, SLuint32 *pCount );`			
Description	Returns the number of children (nodes, streams, etc.) within the current scope.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pCount	[out]	Number of children.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Child count is determined by the metadata traversal mode [see section 9.2.23]. If the mode is set to SL_METADATATRAVERSALMODE_ALL, GetChildCount() will always return 0. If the mode is set to SL_METADATATRAVERSALMODE_NODE, GetChildCount() will return the number of nodes within the current scope. For example, in a Mobile XMF file with one SMF node and one Mobile DLS node, GetChildCount() will return 2 from the root.		
See also	SetMode()		

GetChildMIMETypeSize

```
SLresult (*GetChildMIMETypeSize) (
   SLMetadataTraversalItf self,
   SLuint32 index,
   SLu3int2 *pSize
);
```

Description	Returns the size in bytes needed to store the MIME type of a child.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	Child index. Range is [0, GetChildCount()).
	pSize	[out]	Size of the MIME type in bytes.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	GetChildCount()		

GetChildInfo

```
SLresult (*GetChildInfo) (
   SLMetadataTraversalItf self,
   SLuint32 index,
   SLint32 *pNodeID,
   SLuint32 *pType,
   SLuint32 size,
   SLchar *pMimeType
);
```

Description	Returns information about a child.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	Child index. Range is [0, GetChildCount()).
	pNodeID	[out]	Unique identification number of the child.
	pType	[out]	Node type. See SL_NODETYPE macros [see section 9.2.29].
	size	[in]	Size of the memory block passed as mimeType. Range is (0, max GetChildMIMETypeSize()].
	pMimeType	[out]	Address to store the MIME type.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	To ignore MIME type, set size to 0 and pMimeType to NULL.		
See also	GetChildCount()		

SetActiveNode			
`SLresult (*SetActiveNode) ( SLMetadataTraversalItf self, SLuint32 index );`			
Description	Sets the scope to a child index.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`index`	[in]	Child index. Range is special (see below).

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	`SetActiveNode()` causes the current scope to descend or ascend to the given index. To descend, set `index` to [0, `GetChildCount()`). To ascend to the parent scope, set `index` to `SL_NODE_PARENT`. Calling `SetActiveNode()` with `index` set to `SL_NODE_PARENT` will return `SL_RESULT_PARAMETER_INVALID` if the active node is root.		
See also	`GetChildCount()`, `SL_NODE_PARENT` [see section 9.2.28]		

8.29 SLMIDIMessageItf

Description

The MIDI message interface is used in order to send MIDI messages directly to a MIDI-based player, and to set MIDI message and meta-event callbacks. It is used primarily to determine the state of the internal MIDI synthesizer at runtime by setting callbacks that report that state, as well as to set the state via the SendMessage method.

This interface is supported on the MIDI Player [see section 7.8] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_MIDIMESSAGE;

struct SLMIDIMessageItf_;
typedef const struct SLMIDIMessageItf * const * SLMIDIMessageItf;

struct SLMIDIMessageItf_ {
   SLresult (*SendMessage) (
         SLMIDIMessageItf self,
         const SLuint8 *pData,
         SLuint32 length
   );
   SLresult (*RegisterMetaEventCallback) (
         SLMIDIMessageItf self,
         slMetaEventCallback callback,
         void *pContext
   );
   SLresult (*RegisterMIDIMessageCallback) (
         SLMIDIMessageItf self,
         slMIDIMessageCallback callback,
         void *pContext
   );
   SLresult (*AddMIDIMessageCallbackFilter) (
         SLMIDIMessageItf self,
         SLuint32 messageType
   );
   SLresult (*ClearMIDIMessageCallbackFilter) (
         SLMIDIMessageItf self
   );
};
```

Interface ID

ddf4a820-ddd6-11db-b174-0002a5d5c51b

Callbacks

slMetaEventCallback

```
typedef void (SLAPIENTRY *slMetaEventCallback) (
   SLMIDIMessageItf caller,
   void *pContext,
   SLuint8 type,
   SLuint32 length,
   const SLuint8 *pData,
   SLuint32 tick,
   SLuint16 track
);
```

Description	Executes when a MIDI-based player encounters an SMF meta-event.		
Parameters	caller	[in]	Interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	type	[in]	Type of the meta-event, as specified in MIDI specification. Range is [0,127].
	length	[in]	Length of the meta-event data.
	pData	[in]	Address of an array of bytes containing the meta-event data (may be NULL if length is 0).
	tick	[in]	SMF tick at which the meta-event was encountered.
	track	[in]	SMF track on which the meta-event was encountered.
Comments	slMetaEventCallback returns the address of the entire meta-event (not including the 0xFF identifier, the type byte, or the size VLQ), the MIDI tick at which the meta-event was found, and the Standard MIDI File track number. For example, the data of a Set Tempo meta-event for 500,000 microseconds per quarter note (120 BPM) would be represented by the following byte sequence: 0x07 0xA1 0x20, since 0x7A120 = 500000.		
See also	RegisterMetaEventCallback()		

slMIDIMessageCallback

```
typedef void (SLAPIENTRY *slMIDIMessageCallback) (
   SLMIDIMessageItf caller,
   void *pContext,
   SLuint8 statusByte,
   SLuint32 length,
   const SLuint8 *pData,
   SLuint32 tick,
   SLuint16 track
);
```

Description	Executes when a MIDI-based player encounters a MIDI message.		
Parameters	`caller`	[in]	Interface instantiation on which the callback was registered.
	`pContext`	[in]	User context data that is supplied when the callback method is registered.
	`statusByte`	[in]	Status byte of the MIDI message. Range is [0x80, 0xFF].
	`length`	[in]	Length of the MIDI message, not including the status byte. `length` must be greater than 0.
	`pData`	[in]	Address of an array of bytes containing the MIDI message data bytes.
	`tick`	[in]	SMF tick at which the MIDI message was encountered.
	`track`	[in]	SMF track on which the MIDI message was encountered.
Comments	`slMIDIMessageCallback` returns the status byte of the MIDI message, the address of the data byte(s) of the MIDI message (not including the status byte), the MIDI tick at which the MIDI message was found, and, if applicable, the Standard MIDI file track number. For example, a Note On message on channel 3 for note 64 with velocity 96 (0x93 0x40 0x60) would be represented by the following data byte sequence: 0x40 0x60. It would also return the status byte as 0x93. Note: if the origin of the MIDI message is a MIDI buffer queue, track will be set to 0. Otherwise, track will be set to the SMF track on which the MIDI message was encountered (zero-based, that is, when the track parameter is 0, the message was contained in the first track in the SMF). Note: `slMIDIMessageCallback` passes the status byte separately to ensure that MIDI messages that use running status are properly identified.		
See also	`RegisterMIDIMessageCallback()`		

Methods

SendMessage

```
SLresult (*SendMessage) (
   SLMIDIMessageItf self,
   const SLuint8 *pData,
   SLuint32 length
);
```

Description	Sends a MIDI message to a MIDI-based player.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pData`	[in]	Address of an array of bytes containing the MIDI message.
	`length`	[in]	Length of the MIDI message data. `length` must be greater than 0.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE SL_RESULT_CONTENT_CORRUPTED		
Comments	See the MIDI 1.0 Detailed Specification [MIDI] for details on the format of MIDI messages, and the Standard MIDI Files 1.0 specification for details on the format of SMF data including timing information and meta events. `SendMessage()` must begin with a valid MIDI status byte. Note: `SendMessage()` does not support sending SMF meta-events. The set of supported MIDI messages includes those specified in the SP-MIDI Device 5-24 Note Profile for 3GPP and the Mobile DLS Specification. MIDI player support for any additional MIDI messages is optional, and specific to the implementation. The play state does not affect controlling the MIDI Player with this method.		
See also	None.		

RegisterMetaEventCallback

```
SLresult (*RegisterMetaEventCallback) (
   SLMIDIMessageItf self,
   slMetaEventCallback callback,
   void *pContext
);
```

Description	Sets or clears a MIDI-based player's SMF meta-event callback.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	callback	[in]	Address of the meta-event callback.
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	See the Standard MIDI Files 1.0 specification [MIDI] for details on the format of SMF meta-events.		
See also	slMetaEventCallback		

RegisterMIDIMessageCallback

```
SLresult (*RegisterMIDIMessageCallback) (
    SLMIDIMessageItf self,
    slMIDIMessageCallback callback,
    void *pContext
);
```

Description	Sets or clears a MIDI-based player's MIDI message callback.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	callback	[in]	Address of the MIDI message callback.
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	It is necessary to add specific MIDI message types to the MIDI message filter in order for the slMIDIMessageCallback to execute. See the MIDI 1.0 Detailed Specification [MIDI] for details on the format of MIDI messages.		
See also	slMIDIMessageCallback(), AddMIDIMessageCallbackFilter(), ClearMIDIMessageCallbackFilter ()		

AddMIDIMessageCallbackFilter

```
SLresult (*AddMIDIMessageCallbackFilter) (
   SLMIDIMessageItf self,
   SLuint32 messageType
);
```

Description	Adds a MIDI message type to the player's MIDI message callback filter.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`messageType`	[in]	MIDI message type to filter. Must be one of the `SL_MIDIMESSAGETYPE` macros.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	The MIDI message callback filter is an additive opt-in filter that adds MIDI message types to a list of messages that are used for determining whether or not to execute the `slMIDIMessageCallback`. If no message types are added, `slMIDIMessageCallback` will not execute. `AddMIDIMessageCallbackFilter()` may be called at any time.		
See also	`slMIDIMessageCallback`, `ClearMIDIMessageCallbackFilter()`, `RegisterMIDIMessageCallback()`, `SL_MIDIMESSAGETYPE`		

ClearMIDIMessageCallbackFilter

```
SLresult (*ClearMIDIMessageCallbackFilter) (
   SLMIDIMessageItf self
);
```

Description	Clears the player's MIDI message callback filter.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
Return value	The return value can be one of the following: `SL_RESULT_SUCCESS`		
Comments	`ClearMIDIMessageCallbackFilter()` may be called at any time.		
See also	`slMIDIMessageCallback`, `AddMIDIMessageCallbackFilter()`, `RegisterMIDIMessageCallback()`		

8.30 SLMIDIMuteSoloItf

Description

The interface to mute and solo MIDI channels and tracks. It also returns the MIDI track count.

Muting a MIDI channel or track silences it until it is unmuted. Soloing a channel or track silences all other non-soloed channels or tracks. While silenced, channels and tracks continue to process MIDI messages, but simply cease to contribute any Note On messages to the player's output.

It is possible to solo more than one channel or track at a time. In such circumstances, only those channels or tracks that are soloed contribute to the player's output. It is also possible to mix muting and soloing on a channel or track. In this case, a channel or track will be heard if and only if it is not muted and it is soloed or no other channel or track is soloed.

Finally, it is possible to mix channel and track soloing and muting. Because tracks can play to multiple channels, muting a track will silence whatever contribution it would make to that channel.

Example:

Assume the following SP-MIDI file:

- Track 1 plays data to channels 0, 1, and 2
- Track 2 plays data to channels 2 and 3
- Track 3 plays data to channels 3

Muting channel 2 will silence the channel, implicitly silencing the contribution track 1 and track 2 make to that channel (track 3's output is unaffected).

Soloing channel 3 will silence all other channels, affecting the contribution of all tracks except for track 3, which is unaffected because its entire contribution to the output is to the soloed channel. Adding a solo to channel 2 will silence all channels other than 2 and 3, meaning that only track 1 is affected (its contribution to channels 0 and 1 are silenced, channel 2 continues to play).

Muting track 2 silences its contribution to channels 2 and 3, but allows tracks 1 and 3 to continue unaffected.

Soloing track 2 silences the contribution all other tracks make. Adding a solo to track 3 allows track 2 and track 3 to play unaltered and to silence track 1.

Soloing Track 2 and soloing channel 0 effectively silences all playback; the effect is to allow only track 2 to contribute to the player's output, and only on channel 0, which it does not use. In this case, adding a solo to track 1 would allow track 1 to play on channel 0 alone. Alternately, adding a solo to channel 2 would allow track 2 to play channel 2.

This interface is supported on the MIDI Player [see section 7.8] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_MIDIMUTESOLO;

struct SLMIDIMuteSoloItf_;
typedef const struct SLMIDIMuteSoloItf_ * const * SLMIDIMuteSoloItf;

struct SLMIDIMuteSoloItf_ {
   SLresult (*SetChannelMute) (
         SLMIDIMuteSoloItf self,
         SLuint8 channel,
         SLboolean mute
   );
   SLresult (*GetChannelMute) (
         SLMIDIMuteSoloItf self,
         SLuint8 channel,
         SLboolean *pMute
   );
   SLresult (*SetChannelSolo) (
         SLMIDIMuteSoloItf self,
         SLuint8 channel,
         SLboolean solo
   );
   SLresult (*GetChannelSolo) (
         SLMIDIMuteSoloItf self,
         SLuint8 channel,
         SLboolean *pSolo
   );
   SLresult (*GetTrackCount) (
         SLMIDIMuteSoloItf self,
         SLuint16 *pCount
   );
   SLresult (*SetTrackMute) (
         SLMIDIMuteSoloItf self,
         SLuint16 track,
         SLboolean mute
   );
   SLresult (*GetTrackMute) (
         SLMIDIMuteSoloItf self,
         SLuint16 track,
         SLboolean *pMute
   );
```

```
    SLresult (*SetTrackSolo) (
          SLMIDIMuteSoloItf self,
          SLuint16 track,
          SLboolean solo
    );
    SLresult (*GetTrackSolo) (
          SLMIDIMuteSoloItf self,
          SLuint16 track,
          SLboolean *pSolo
    );
};
```

Interface ID

039eaf80-ddd7-11db-9a02-0002a5d5c51b

Methods

SetChannelMute			
`SLresult (*SetChannelMute) ( SLMIDIMuteSoloItf self, SLuint8 channel, SLboolean mute );`			
Description	Mutes or unmutes a MIDI channel on a MIDI-based player.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a SLMIDIMuteSoloItf interface.
	channel	[in]	MIDI channel. Range is [0, 15].
	mute	[in]	Boolean indicating whether to mute or unmute the MIDI channel. SL_BOOLEAN_TRUE specifies that the MIDI channel should be muted; SL_BOOLEAN_FALSE specifies that the MIDI channel should be unmuted.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE		
Comments	None.		
See also	None.		

GetChannelMute

```
SLresult (*GetChannelMute) (
    SLMIDIMuteSoloItf self,
    SLuint8 channel,
    SLboolean *pMute
);
```

Description	Returns whether a MIDI channel on a MIDI-based player is muted or unmuted.		
Pre-conditions	None		
Parameters	`self`	[in]	Pointer to a `SLMIDIMuteSoloItf` interface.
	`channel`	[in]	Address to store MIDI channel. Range is [0, 15].
	`pMute`	[out]	Address to store a Boolean that indicates whether a MIDI channel is muted. This must be non-`NULL`. `SL_BOOLEAN_TRUE` indicates that the MIDI channel is muted; `SL_BOOLEAN_FALSE` indicates that the MIDI channel is unmuted.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		
See also	None.		

SetChannelSolo			
```
SLresult (*SetChannelSolo) (
   SLMIDIMuteSoloItf self,
   SLuint8 channel,
   SLboolean solo
);
```
Description	Solos or unsolos a MIDI channel on a MIDI-based player.		
Pre-conditions	None.		
Parameters	`self`	[in]	Pointer to a `SLMIDIMuteSoloItf` interface.
	`channel`	[in]	MIDI channel. Range is [0, 15].
	`solo`	[in]	Boolean indicating whether to solo or unsolo the MIDI channel. `SL_BOOLEAN_TRUE` specifies that the MIDI channel should be soloed; `SL_BOOLEAN_FALSE` specifies that the MIDI channel should be unsoloed.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_MEMORY_FAILURE`		

Comments	None.		
See also	None.		

GetChannelSolo

```
SLresult (*GetChannelSolo) (
   SLMIDIMuteSoloItf self,
   SLuint8 channel,
   SLboolean *pSolo
);
```

Description	Returns whether a MIDI channel on a MIDI-based player is soloed or unsoloed.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a SLMIDIMuteSoloItf interface.
	channel	[in]	Address to store MIDI channel. Range is [0, 15].
	pSolo	[out]	Address to store a Boolean indicating whether a MIDI channel is soloed. This must be non-NULL. SL_BOOLEAN_TRUE specifies that the MIDI channel is soloed; SL_BOOLEAN_FALSE specifies that the MIDI channel is unsoloed.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

GetTrackCount

```
SLresult (*GetTrackCount) (
    SLMIDIMuteSoloItf self,
    SLuint16 *pCount
);
```

Description	Returns the number of MIDI tracks in a MIDI-based player's SMF data.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a SLMIDIMuteSoloItf interface.
	pCount	[out]	Address to store the number of MIDI tracks. This must be non-NULL. Range is [1 to 65535].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	MIDI buffer queues are treated as SMF type 0 files, and thus always return 1.		
See also	None.		

SetTrackMute

```
SLresult (*SetTrackMute) (
   SLMIDIMuteSoloItf self,
   SLuint16 track,
   SLboolean mute
);
```

Description	Mutes or unmutes a MIDI track on a MIDI-based player.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a SLMIDIMuteSoloItf interface.
	track	[in]	MIDI track. Range is [0, SLMIDIMuteSoloItf::GetTrackCount()).
	mute	[in]	Boolean indicating whether to mute or unmute the MIDI track. SL_BOOLEAN_TRUE specifies that the MIDI track should be muted; SL_BOOLEAN_SL_BOOLEAN_FALSE specifies that the MIDI track should be unmuted.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE		
Comments	None.		
See also	None.		

GetTrackMute

```
SLresult (*GetTrackMute) (
   SLMIDIMuteSoloItf self,
   SLuint16 track,
   SLboolean *pMute
);
```

Description	Returns whether a MIDI track on a MIDI-based player is muted or unmuted.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a SLMIDIMuteSoloItf interface.
	track	[in]	Address to store MIDI track. Range is [0, SLMIDIMuteSoloItf::GetTrackCount()).
	pMute	[out]	Address to store a Boolean indicating whether a MIDI track is muted. This must be non-NULL. SL_BOOLEAN_TRUE specifies that the MIDI track is muted; SL_BOOLEAN_FALSE specifies that the MIDI track is unmuted.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

SetTrackSolo

```
SLresult (*SetTrackSolo) (
    SLMIDIMuteSoloItf self,
    SLuint16 track,
    SLboolean solo
);
```

Description	Solos or unsolos a MIDI track on a MIDI-based player.		
Pre-conditions	None.		
Parameters	`self`	[in]	Pointer to a `SLMIDIMuteSoloItf` interface.
	`track`	[in]	MIDI track. Range is [0, `SLMIDIMuteSoloItf::GetTrackCount()`).
	`solo`	[in]	Boolean indicating whether to solo or unsolo the MIDI track. SL_BOOLEAN_TRUE specifies that the MIDI track should be soloed; SL_BOOLEAN_FALSE specifies that the MIDI track should be unsoloed.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE		
Comments	None.		
See also	None.		

GetTrackSolo

```
SLresult (*GetTrackSolo) (
   SLMIDIMuteSoloItf self,
   SLuint16 track,
   SLboolean *pSolo
);
```

Description	Returns whether a MIDI track on a MIDI-based player is soloed or unsoloed.		
Pre-conditions	None.		
Parameters	`self`	[in]	Pointer to a `SLMIDIMuteSoloItf` interface.
	`track`	[in]	Address to store MIDI track. [0, `SLMIDIMuteSoloItf::GetTrackCount()`).
	`pSolo`	[out]	Address to store a Boolean indicating whether a MIDI track is soloed. This must be non-`NULL`. `SL_BOOLEAN_TRUE` specifies that the MIDI track is soloed; `SL_BOOLEAN_FALSE` specifies that the MIDI track is unsoloed.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		
See also	None.		

8.31 SLMIDITempoItf

Description

Interface for interacting with the MIDI data's tempo.

This interface is supported on the MIDI Player [see section 7.8] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_MIDITEMPO;

struct SLMIDITempoItf_;
typedef const struct SLMIDITempoItf_ * const * SLMIDITempoItf;

struct SLMIDITempoItf_ {
   SLresult (*SetTicksPerQuarterNote) (
         SLMIDITempoItf self,
         SLuint32 tpqn
   );
   SLresult (*GetTicksPerQuarterNote) (
         SLMIDITempoItf self,
         SLuint32 *pTpqn
   );
   SLresult (*SetMicrosecondsPerQuarterNote) (
         SLMIDITempoItf self,
         SLmicrosecond uspqn
   );
   SLresult (*GetMicrosecondsPerQuarterNote) (
         SLMIDITempoItf self,
         SLmicrosecond *uspqn
   );
};
```

Interface ID

1f347400-ddd7-11db-a7ce-0002a5d5c51b

Defaults

- 96 ticks per quarter note.
- 500,000 microseconds per quarter note (120 BPM).

Methods

SetTicksPerQuarterNote

```
SLresult (*SetTicksPerQuarterNote) (
   SLMIDITempoItf self,
   SLuint32 tpqn
);
```

Description	Sets a MIDI-based player's time in ticks per quarter note.		
Pre-conditions	Playback must be stopped.		
Parameters	`self`	[in]	Pointer to a `MIDITempoItf` interface.
	`tpqn`	[in]	Ticks per quarter note. Range is [1, 32767].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	This method is intended for setting the tick resolution when working with a buffer queue source on a MIDI player. When playing back MIDI files, the file itself contains this information. Note: ticks per quarter note is also known as ticks per beat and pulses per quarter note.		
See also	None.		

GetTicksPerQuarterNote

```
SLresult (*GetTicksPerQuarterNote) (
    SLMIDITempoItf self,
    SLuint32 *pTpqn
);
```

Description	Returns a MIDI-based player's delta-time in ticks per quarter note.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a MIDITempoItf interface.
	pTpqn	[out]	Address to store ticks per quarter note. This must be non-NULL. Range is [0, 32767], where 0 has a special meaning (see comments).
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Some SMF files define the tick resolution in terms other than ticks per quarter note (e.g. in terms of SMPTE frames). In such cases, the returned value will be 0. Note: ticks per quarter note is also known as ticks per beat and pulses per quarter note.		
See also	None.		

SetMicrosecondsPerQuarterNote

```
SLresult (*SetMicrosecondsPerQuarterNote) (
    SLMIDITempoItf self,
    SLmicrosecond uspqn
);
```

Description	Sets a MIDI-based player's tempo in microseconds per quarter note.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a `SLMIDITempoItf` interface.
	uspqn	[in]	Tempo in microseconds per quarter note. Range is [0, 16,777,215].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Example: 500,000 microseconds per quarter note yields 120 beats per minute. Note: The player's SMF metadata may set the tempo independently, thus overriding any value that `SetMicrosecondsPerQuarterNote()` may set.		
See also	None.		

GetMicrosecondsPerQuarterNote

```
SLresult (*GetMicrosecondsPerQuarterNote) (
    SLMIDITempoItf self,
    SLmicrosecond *uspqn
);
```

Description	Returns a MIDI-based player's tempo in microseconds per quarter note.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a `SLMIDITempoItf` interface.
	uspqn	[out]	Address to store tempo in microseconds per quarter note. This must be non-`NULL`. Range is [0, 16,777,215].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Example: 500,000 microseconds per quarter note yields 120 beats per minute.		
See also	None.		

8.32 SLMIDITimeItf

Description

Interface for interacting with the MIDI data in time (ticks). This interface is not available on MIDI-based players created using `SLDataLocator_MIDIBufferQueue`.

This interface is supported on the MIDI Player [see section 7.8] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_MIDITIME;

struct SLMIDITimeItf_;
typedef const struct SLMIDITimeItf_ * const * SLMIDITimeItf;

struct SLMIDITimeItf_ {
   SLresult (*GetDuration) (
         SLMIDITimeItf self,
         SLuint32 *pDuration
   );
   SLresult (*SetPosition) (
         SLMIDITimeItf self,
         SLuint32 position
   );
   SLresult (*GetPosition) (
         SLMIDITimeItf self,
         SLuint32 *pPosition
   );
   SLresult (*SetLoopPoints) (
         SLMIDITimeItf self,
         SLuint32 startTick,
         SLuint32 numTicks
   );
   SLresult (*GetLoopPoints) (
         SLMIDITimeItf self,
         SLuint32 *pStartTick,
         SLuint32 *pNumTicks
   );
};
```

Interface ID

3da51de0-ddd7-11db-af70-0002a5d5c51b

Methods

GetDuration			
`SLresult (*GetDuration) ( SLMIDITimeItf self, SLuint32 *pDuration );`			
Description	Returns the duration of a MIDI-based player in MIDI ticks.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a SLMIDITimeItf interface.
	pDuration	[out]	Address to store the MIDI tick duration. This must be non-NULL. pDuration must be greater than 0.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	SLMIDITempoItf::GetTicksPerQuarterNote().		

```
SLresult (*GetDuration) (
    SLMIDITimeItf self,
    SLuint32 *pDuration
);
```

SetPosition			
`SLresult (*SetPosition) ( SLMIDITimeItf self, SLuint32 position );`			
Description	Sets a MIDI-based player's current position in MIDI ticks.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a SLMIDITimeItf interface.
	position	[in]	MIDI tick position. Range is [0, SLMIDITimeItf::GetDuration()).
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	SLMIDITempoItf::GetTicksPerQuarterNote().		

```
SLresult (*SetPosition) (
    SLMIDITimeItf self,
    SLuint32 position
);
```

GetPosition

```
SLresult (*GetPosition) (
    SLMIDITimeItf self,
    SLuint32 *pPosition
);
```

Description	Returns a MIDI-based player's current position in MIDI ticks.		
Pre-conditions	None.		
Parameters	self	[in]	Pointer to a SLMIDITimeItf interface.
	pPosition	[out]	Address to store the MIDI tick position. This must be non-NULL. Range is [0, SLMIDITimeItf::GetDuration()).
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	SLMIDITempoItf::GetTicksPerQuarterNote().		

SetLoopPoints

```
SLresult (*SetLoopPoints) (
    SLMIDITimeItf self,
    SLuint32 startTick,
    SLuint32 numTicks
);
```

Description	Sets a MIDI-based player's loop points in MIDI ticks.		
Pre-conditions	The player must be in the SL_PLAYSTATE_STOPPED state.		
Parameters	self	[in]	Pointer to a SLMIDITimeItf interface.
	startTick	[in]	MIDI tick position of loop start. Range is [0, SLMIDITimeItf::GetDuration()).
	numTicks	[in]	Number of MIDI ticks in the loop. Range is [0, SLMIDITimeItf::GetDuration() - startTick], where 0 is a special value, designating "no looping."
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	Note: setting loop points in MIDI files may cause unpredictable audio output if the MIDI data is not known. This is due to a number of reasons, including the possibility of looping back after a note message on has been sent, but before the corresponding Note Off was sent, or looping back to MIDI data that plays properly only when a specific set of controllers are set.		
See also	SLMIDITempoItf::GetTicksPerQuarterNote().		

GetLoopPoints

```
SLresult (*GetLoopPoints) (
    SLMIDITimeItf self,
    SLuint32 *pStartTick,
    SLuint32 *pNumTicks
);
```

Description	Returns a MIDI-based player's loop points in MIDI ticks.		
Pre-conditions	None.		
Parameters	`self`	[in]	Pointer to a `SLMIDITimeItf` interface.
	`pStartTick`	[out]	Address to store the MIDI tick position of loop start. This must be non-`NULL`. Range is [0, `SLMIDITimeItf::GetDuration()`).
	`pNumTicks`	[out]	Address to store the number of MIDI ticks in the loop. This must be non-`NULL`. Range is [0, `SLMIDITimeItf::GetDuration() - startTick`], where 0 is a special value, designating "no looping."
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	GetLoopPoints returns `startTick` as 0 and `numTicks` as `SLMIDITimeItf::GetDuration()` if not previously set.		
See also	`SLMIDITempoItf::GetTicksPerQuarterNote()`.		

8.33 SLMuteSoloItf

Description

This interface exposes controls for selecting which of the player's channels are heard and silenced.

The following restriction is placed on this interface:

- This interface cannot be exposed on a player whose audio format is mono.

This interface is supported on the Audio Player [see section 7.2] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_MUTESOLO;

struct SLMuteSoloItf_;
typedef const struct SLMuteSoloItf_ * const * SLMuteSoloItf;

struct SLMuteSoloItf_ {
   SLresult (*SetChannelMute) (
         SLMuteSoloItf self,
         SLuint8 chan,
         SLboolean mute
   );
   SLresult (*GetChannelMute) (
         SLMuteSoloItf self,
         SLuint8 chan,
         SLboolean *pMute
   );
   SLresult (*SetChannelSolo) (
         SLMuteSoloItf self,
         SLuint8 chan,
         SLboolean solo
   );
   SLresult (*GetChannelSolo) (
         SLMuteSoloItf self,
         SLuint8 chan,
         SLboolean *pSolo
   );
   SLresult (*GetNumChannels) (
         SLMuteSoloItf self,
         SLuint8 *pNumChannels
   );
};
```

Interface ID

5a28ebe0-ddd7-11db-8220-0002a5d5c51b

Defaults

- No channels muted
- No channels soloed

Methods

SetChannelMute			
`SLresult (*SetChannelMute) ( SLMuteSoloItf self, SLuint8 chan, SLboolean mute );`			
Description	Mutes or unmutes a specified channel of a player.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`chan`	[in]	Channel to mute or unmute. The valid range is [0, n-1], where n is the number of audio channels in player's audio format.
	`mute`	[in]	If set to true, mutes the channel. If set to false, unmutes the channel.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	When a channel is muted it should continue to play silently in synchronization with the other channels.		
See also	`SLVolumeItf::SetMute(), SetChannelSolo()`		

GetChannelMute

```
SLresult (*GetChannelMute) (
   SLMuteSoloItf self,
   SLuint8 chan,
   SLboolean *pMute
);
```

Description	Retrieves the mute status of a specified channel of a player.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	chan	[in]	Channel to query the mute state. The valid range is [0, n-1], where n is the number of audio channels in player's audio format.
	pMute	[out]	Pointer to a location to receive the mute status of the specified channel. If set to true, the channel is muted. If set to false, the channel is not muted. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This method only returns the per-channel mute setting and is independent of any global mute or per-channel soloing that may be enabled.		
See also	SLVolumeItf::SetMute()		

SetChannelSolo

```
SLresult (*SetChannelSolo) (
    SLMuteSoloItf self,
    SLuint8 chan,
    SLboolean solo
);
```

Description	Enables or disables soloing of a specified channel of a player.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`chan`	[in]	Channel to solo or unsolo. The valid range is [0, n-1], where n is the number of audio channels in player's audio format.
	`solo`	[in]	If set to true, the channel is soloed; all non-soloed channels are silenced. If set to false, disables any soloing currently enabled for the specified channel.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	When any channel is soloed, all non-soloed channels should continue to play silently in synchronization with the soloed channels. If a channel is both muted and soloed (using `SetChannelMute()`), the channel will be silent until the channel is unmuted. If no channels are soloed, all unmuted channels are heard.		
See also	`SetChannelMute()`		

GetChannelSolo

```
SLresult (*GetChannelSolo) (
   SLMuteSoloItf self,
   SLuint8 chan,
   SLboolean *pSolo
);
```

Description	Retrieves the soloed state of a specified channel of a player.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	chan	[in]	Channel to query the solo state. The valid range is [0, n-1], where n is the number of audio channels in player's audio format.
	pSolo	[out]	Pointer to a location to receive the solo status of the specified channel. If set to true, the channel is soloed. If set to false, the channel is not soloed. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

GetNumChannels

```
SLresult (*GetNumChannels) (
   SLMuteSoloItf self,
   SLuint8 *pNumChannels
);
```

Description	Retrieves the number of audio channels contained in the player's audio format		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pNumChannels	[out]	Pointer to a location to receive the number of audio channels contained in the player's audio format. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

8.34 SLObjectItf

Description

The `SLObjectItf` interface provides essential utility methods for all objects. Such functionality includes the destruction of the object, realization and recovery, acquisition of interface pointers, a callback for runtime errors, and asynchronous operation termination.

A maximum of one asynchronous operation may be performed by an object at any given time. Trying to invoke an asynchronous operation when an object is already processing an asynchronous call is equivalent to aborting the first operation, then invoking the second one.

`SLObjectItf` is an implicit interface of all object types and is automatically available upon creation of every object.

Please refer to section 3.1.1 for details on the object states.

This interface is supported on all objects [see section 7].

See Appendix B: and Appendix C: for examples using this interface.

Priority

This interface exposes a control for setting an object's priority relative to the other objects under control of the same instance of the engine. This priority provides a hint that the implementation can use when there is resource contention between objects.

Given resource contention between objects, an implementation will give preference to the object with the highest priority. This may imply that the implementation takes resources from one object to give to another if the two objects are competing for the same resources and the latter has higher priority. Given objects of identical priority competing for resources, the implementation steals the resources from the object that acquired them earlier to give to the object that requested them most recently.

Different objects may require entirely different resources. For this reason, it is possible that an object of high priority may have its resources stolen before an object of low priority. For example, a high-priority object may require access to dedicated hardware on the device while the low-priority object does not. If this dedicated hardware is demanded by the system, the resources may need to be stolen from the higher priority object, leaving the low priority object in the Realized state.

Loss of Control

Loss of control of an interface and its associated underlying resource means that the application has lost the ability to set the parameters of the interface and resource, but the interface and resource are otherwise still functioning and available for the application to use.

For example, imagine a system with a maximum of one environmental reverb unit (limited to one due to memory/CPU constraints, or because it is implemented in a hardware DSP). An application configures the environmental reverb for its output mix A. Another higher-priority application configures the environmental reverb for its own output mix B. The OpenSL ES implementation may now signal to the first application that output mix A has lost control of the reverb (it is still applied on A, but with the same parameters as on B).

See the related object event macros (`SL_OBJECT_EVENT_ITF_CONTROL_TAKEN`, `SL_OBJECT_EVENT_ITF_CONTROL_RETURNED` and `SL_OBJECT_EVENT_ITF_PARAMETERS_CHANGED` [see section 9.2.30]) and the error code `SL_RESULT_CONTROL_LOST` [see section 9.2.46] for details.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_OBJECT;

struct SLObjectItf_;
typedef const struct SLObjectItf_ * const * SLObjectItf;

struct SLObjectItf_ {
   SLresult (*Realize) (
         SLObjectItf self,
         SLboolean async
   );
   SLresult (*Resume) (
         SLObjectItf self,
         SLboolean async
   );
   SLresult (*GetState) (
         SLObjectItf self,
         SLuint32 * pState
   );
   SLresult (*GetInterface) (
         SLObjectItf self,
         const SLInterfaceID iid,
         void * pInterface
   );
   SLresult (*RegisterCallback) (
         SLObjectItf self,
         slObjectCallback callback,
         void * pContext
   );
   void (*AbortAsyncOperation) (
         SLObjectItf self
   );
   void (*Destroy) (
         SLObjectItf self
   );
   SLresult (*SetPriority) (
         SLObjectItf self,
         SLuint32 priority
   );
```

```
    SLresult (*GetPriority) (
          SLObjectItf self,
          SLuint32 *pPriority
    );
    SLresult (*SetLossOfControlInterfaces) (
          SLObjectItf self,
          SLuint16 numInterfaces,
          const SLInterfaceID * pInterfaceIDs,
          SLboolean enabled
    );
};
```

Interface ID

ba58c2e0-cb47-11df-bd3b-0800200c9a66

Defaults

The object is in Unrealized state.

No callback is registered.

Priority: SL_PRIORITY_NORMAL

Preemptable by object of same priority that is realized later than this object: SL_BOOLEAN_FALSE

Callbacks

slObjectCallback

```
typedef void (SLAPIENTRY *slObjectCallback) (
   SLObjectItf caller,
   const void * pContext,
   SLuint32 event,
   SLresult result,
   SLuint32 param,
   void * pInterface
);
```

Description	A callback function, notifying of a runtime error, termination of an asynchronous call or change in the object's resource state.		
Parameters	`caller`	[in]	Interface which invoked the callback.
	`pContext`	[in]	User context data that is supplied when the callback method is registered.
	`event`	[in]	One of the `SL_OBJECT_EVENT` macros.
	`result`	[in]	If the event is `SL_OBJECT_EVENT_RUNTIME_ERROR`, `result` contains the error code. If the event is `SL_OBJECT_EVENT_ASYNC_TERMINATION`, `result` contains the asynchronous function return code. For other values of `event`, this parameter should be ignored.
	`param`	[in]	If event is `SL_OBJECT_EVENT_RUNTIME_ERROR`, `SL_OBJECT_EVENT_RESOURCES_LOST` or `SL_OBJECT_EVENT_ASYNC_TERMINATION`, `param` contains the state of the object after the event. For other values of `event`, this parameter should be ignored.
	`pInterface`	[in]	If event is `SL_OBJECT_EVENT_ITF_CONTROL_TAKEN`, `SL_OBJECT_EVENT_ITF_CONTROL_RETURNED` or `SL_OBJECT_EVENT_ITF_PARAMETERS_CHANGED`, `pInterface` contains the interface affected. For other values of `event`, this parameter should be ignored.
Comments	Please note the restrictions applying to operations performed from within callback context, in section 3.3.		
See also	`RegisterCallback()`		

Methods

Realize			
`SLresult (*Realize) ( SLObjectItf self, SLboolean async );`			
Description	Transitions the object from Unrealized state to Realized state, either synchronously or asynchronously.		
Pre-conditions	The object must be in Unrealized state.		
Parameters	self	[in]	Interface self-reference.
	async	[in]	If SL_BOOLEAN_FALSE, the method will block until termination. Otherwise, the method will return SL_RESULT_SUCCESS, and will be executed asynchronously. On termination, the slObjectCallback() will be invoked, if registered, with the SL_OBJECT_EVENT_ASYNC_TERMINATION. The result parameter of the slObjectCallback() will contain the result code of the function. However, if the implementation is unable to initiate the asynchronous call SL_RESULT_RESOURCE_ERROR will be returned.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_RESOURCE_ERROR SL_RESULT_PRECONDITIONS_VIOLATED SL_RESULT_MEMORY_FAILURE SL_RESULT_IO_ERROR SL_RESULT_CONTENT_CORRUPTED SL_RESULT_CONTENT_UNSUPPORTED SL_RESULT_CONTENT_NOT_FOUND SL_RESULT_PERMISSION_DENIED		
Comments	Realizing the object acquires the resources required for its functionality. The operation may fail if insufficient resources are available. In such a case, the application may wait until resources become available and a SL_OBJECT_EVENT_RESOURCES_AVAILABLE event is received, and then retry the realization. Another option is to try and increase the object's priority, thus increasing the likelihood that the object will steal another object's resources.		
See also	Section 3.1.4, SL_OBJECT_EVENT_RESOURCES_AVAILABLE [see section 9.2.30]		

Resume			
`SLresult (*Resume) ( SLObjectItf self, SLboolean async );`			
Description	Transitions the object from Suspended state to Realized state, either synchronously or asynchronously.		
Pre-conditions	The object must be in Suspended state.		
Parameters	self	[in]	Interface self-reference.
	async	[in]	If SL_BOOLEAN_FALSE, the method will block until termination. Otherwise, the method will return SL_RESULT_SUCCESS and will be executed asynchronously. On termination, the slObjectCallback() will be invoked, if registered, with the SL_OBJECT_EVENT_ASYNC_TERMINATION. The result parameter of the slObjectCallback() will contain the result code of the function. However, if the implementation is unable to initiate the asynchronous call SL_RESULT_RESOURCE_ERROR will be returned.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_RESOURCE_ERROR SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	Resuming the object acquires the resources required for its functionality. The operation may fail if insufficient resources are available. In such a case, the application may wait until resources become available and an SL_OBJECT_EVENT_RESOURCES_AVAILABLE event is received, and then retry the resumption. Another option is to try and increase the object's priority, thus increasing the likelihood that the object will steal another object's resources.		
See also	Section 3.1.4, SL_OBJECT_EVENT_RESOURCES_AVAILABLE [see section 9.2.30]		

GetState

```
SLresult (*GetState) (
   SLObjectItf self,
   SLuint32 * pState
);
```

Description	Retrieves the current object state.		
Parameters	self	[in]	Interface self-reference.
	pState	[out]	Pointer to the current state of the object. One of the SL_OBJECT_STATE macros will be written as result [see section 9.2.31]. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
See also	Section 3.1.4		

GetInterface

```
SLresult (*GetInterface) (
   SLObjectItf self,
   const SLInterfaceID iid,
   void * pInterface
);
```

Description	Obtains an interface exposed by the object		
Preconditions	The object is not in the Unrealized state.		
Parameters	`self`	[in]	Interface self-reference.
	`iid`	[in]	The interface type ID.
	`pInterface`	[out]	This should be a non-`NULL` pointer to a variable of the interface's type – for example, if a `SLObjectItf` is retrieved, this parameter should be of type `SLObjectItf *` type.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	If the object does not expose the requested interface type, the return code will be SL_RESULT_FEATURE_UNSUPPORTED.		
See also	Section 3.1.4		

RegisterCallback

```
SLresult (*RegisterCallback) (
   SLObjectItf self,
   slObjectCallback callback,
   void * pContext
);
```

Description	Registers a callback on the object that executes when a runtime error occurs or an asynchronous operation terminates.		
Parameters	`self`	[in]	Interface self-reference.
	`callback`	[in]	Address of the result callback. If `NULL`, the callback is disabled.
	`pContext`	[in]	User context data that is to be returned as part of the callback method.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	The callback will report only runtime errors and results of calls to asynchronous functions.		
See also	`slObjectCallback`		

AbortAsyncOperation

```
void (*AbortAsyncOperation) (
   SLObjectItf self
);
```

Description	Aborts asynchronous call currently processed by the object. This method affects asynchronous calls initiated from `SLObjectItf` or `SLDynamicInterfaceManagementItf` only. If no such call is being processed, the call is ignored. If a callback is registered, it will be invoked, with a `SL_OBJECT_EVENT_ASYNC_TERMINATION` as event and `SL_RESULT_OPERATION_ABORTED` as return code.		
Parameters	`self`	[in]	Interface self-reference.
Return value	None.		
Comments	The method is meant for graceful timeout or user-initiated abortion of asynchronous calls.		

Destroy			
`void (*Destroy) ( SLObjectItf self );`			
Description	Destroys the object.		
Parameters	self	[in]	Interface self-reference.
Return value	None.		
Comments	Destroy implicitly transfers the object through Unrealized state, thus freeing any resources allocated to the object prior to freeing it. All references to interfaces belonging to this object become invalid and may cause undefined behavior if used. All pending asynchronous operations are aborted, as if AbortAsyncOperation() has been called.		

SetPriority			
`SLresult (*SetPriority) ( SLObjectItf self, SLuint32 priority);`			
Description	Set the object's priority.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	priority	[in]	The priority. The valid range for this parameter is [min SLuint32, max SLuint32]. The smaller the number, the higher the priority. Zero is the highest priority.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Although it is possible to set any priority within the specified range, SL_PRIORITY [see section 9.2.39] defines a fixed set of priorities for use with this method.		
See also	None.		

GetPriority

```
SLresult (*GetPriority) (
   struct SLObjectItf self,
   SLuint32 * pPriority
);
```

Description	Gets the object’s priority.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pPriority	[out]	Pointer to a location to receive the object’s priority. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

SetLossOfControlInterfaces

```
SLresult (*SetLossOfControlInterfaces) (
      SLObjectItf self,
      SLuint16 numInterfaces,
      const SLInterfaceID * pInterfaceIDs,
   SLboolean enabled
);
```

Description	Sets/unsets loss of control functionality for a list of interface IDs. The default value of the enabled flag is determined by the global setting (see `SL_ENGINEOPTION_LOSSOFCONTROL 9.2.18`).		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`numInterfaces`	[in]	The length of the `pInterfaceIDs` array.
	`pInterfaceIDs`	[in]	Array of interface IDs representing the interfaces impacted by the enabled flag.
	`enabled`	[in]	If `SL_BOOLEAN_TRUE`, loss of control functionality is enabled for all interfaces represented by `pInterfaceIDs`. If `SL_BOOLEAN_FALSE`, loss of control functionality is disabled for all interfaces represented by `pInterfaceIDs`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	A call to this method overrides the global setting for loss of control functionality for the specified list of interfaces.		

8.35 SLOutputMixItf

Description

SLOutputMixItf is an interface for interacting with an output mix, including querying for the associated destination output devices, registering for the notification of changes to those outputs, and requesting changes to an output mix's associated devices.

This interface is supported on the Output Mix [see section 7.9] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_OUTPUTMIX;

struct SLOutputMixItf_;
typedef const struct SLOutputMixItf_ * const * SLOutputMixItf;

struct SLOutputMixItf_ {
   SLresult (*GetDestinationOutputDeviceIDs) (
         SLOutputMixItf self,
         SLint32 *pNumDevices,
         SLuint32 *pDeviceIDs
   );
   SLresult (*RegisterDeviceChangeCallback) (
         SLOutputMixItf self,
         slMixDeviceChangeCallback callback,
         void *pContext
   );
   SLresult (*ReRoute) (
         SLOutputMixItf self,
         SLint32 numOutputDevices,
         SLuint32 *pOutputDeviceIDs
   );
};
```

Interface ID

97750f60-ddd7-11db-92b1-0002a5d5c51b

Defaults

An output mix defaults to device ID values specific to the implementation.

Callbacks

slMixDeviceChangeCallback			
`typedef void (SLAPIENTRY *slMixDeviceChangeCallback) ( SLOutputMixItf caller, void *pContext );`			
Description	Executes whenever an output mix changes its set of destination output devices. Upon this notification, the application may query for the new set of devices via the `SLOutputMixItf` interface.		
Parameters	`caller`	[in]	Interface on which this callback was registered.
	`pContext`	[in]	User context data that is supplied when the callback method is registered.
Comments	none		
See Also	`RegisterDeviceChangeCallback()`		

```
typedef void (SLAPIENTRY *slMixDeviceChangeCallback) (
   SLOutputMixItf caller,
   void *pContext
);
```

Methods

GetDestinationOutputDeviceIDs

```
SLresult (*GetDestinationOutputDeviceIDs) (
   SLOutputMixItf self,
   SLint32 *pNumDevices,
   SLuint32 *pDeviceIDs
);
```

Description	Retrieves the device IDs of the destination output devices currently associated with the output mix.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pNumDevices	[in/out]	As an input, specifies the length of the pDeviceIDs array (ignored if pDeviceIDs is NULL). As an output, specifies the number of destination output device IDs associated with the output mix.
	pDeviceIDs	[out]	Populated by the call with the list of deviceIDs (provided that pNumDevices is equal to or greater than the number of actual device IDs). If pNumDevices is less than the number of actual device IDs, the error code SL_RESULT_BUFFER_INSUFFICIENT is returned. Note: IDs may include SL_DEFAULTDEVICEID_AUDIOOUTPUT.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_BUFFER_INSUFFICIENT		
Comments	None.		
See also	None.		

RegisterDeviceChangeCallback

```
SLresult (*RegisterDeviceChangeCallback) (
   SLOutputMixItf self,
   slMixDeviceChangeCallback callback
   void * pContext,
);
```

Description	Registers a callback to notify the application when there are changes to the device IDs associated with the output mix.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	callback	[in]	Callback to receive the changes in device IDs associated with the output mix.
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

ReRoute

```
SLresult (*ReRoute)(
  SLOutputMixItf self,
  SLint32 numOutputDevices,
  SLuint32 *pOutputDeviceIDs
);
```

Description	Requests a change to the specified set of output devices on an output mix.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	numOutputDevices	[in]	Number of output devices specified.
	pOutputDeviceIDs	[in]	List of the devices specified. (Note: IDs may include SL_DEFAULTDEVICEID_AUDIOOUTPUT)
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This method simply requests for a change in routing. The implementation may choose not to fulfill the request. If it does not fulfill the request, the method returns SL_RESULT_FEATURE_UNSUPPORTED. **PROFILE NOTE** In the Phone profile, audio can be routed to multiple simultaneous outputs, provided the underlying implementation supports such routing. For implementations that do not support such routing, this method returns *SL_FEATURE_UNSUPPORTED.*		

8.36 SLPitchItf

Description

The `SLPitchItf` interface controls a pitch shift applied to a sound. The pitch shift is specified in permilles:

A value of 1000 ‰ indicates no change in pitch.

A value of 500 ‰ indicates causing a pitch shift of –12 semitones (one octave decrease).

A value of 2000 ‰ indicates a pitch shift of 12 semitones (one octave increase).

Changing the pitch with this interface does not change the playback rate.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_PITCH;

struct SLPitchItf;
typedef const struct SLPitchItf_ * const * SLPitchItf;

struct SLPitchItf_ {
   SLresult (*SetPitch) (
         SLPitchItf self,
         SLpermille pitch
   );
   SLresult (*GetPitch) (
         SLPitchItf self,
         SLpermille *pPitch
   );
   SLresult (*GetPitchCapabilities) (
         SLPitchItf self,
         SLpermille *pMinPitch,
         SLpermille *pMaxPitch
   );
};
```

Interface ID

c7e8ee00-ddd7-11db-a42c-0002a5d5c51b

Defaults

Pitch: 1000 ‰

Methods

SetPitch

```
SLresult (*SetPitch) (
    SLPitchItf self,
    SLpermille pitch
);
```

Description	Sets a player's pitch shift.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pitch`	[in]	The pitch shift factor in permille. The range supported by this parameter is both implementation and content-dependent and can be determined using the `GetPitchCapabilities()` method.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	As the Doppler effect may be implemented internally as a pitch shift and the maximum pitch shift may be capped, in some situations, such as when the pitch shift due to Doppler exceeds the device capabilities, this method may appear to have no audible effect, even though the method succeeded. In this case, the effect will be audible when the amount of Doppler is reduced.		
See also	None.		

GetPitch

```
SLresult (*GetPitch) (
   SLPitchItf self,
   SLpermille *pPitch,
);
```

Description	Gets the player's current pitch shift.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pPitch	[out]	Pointer to a location to receive the player's pitch shift in permille. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

GetPitchCapabilities

```
SLresult (*GetPitchCapabilities) (
SLPitchItf self,
SLpermille *pMinPitch,
SLpermille *pMaxPitch
);
```

Description	Retrieves the player's pitch shifting capabilities.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMinPitch	[out]	Pointer to a location to receive the minimum pitch shift supported for the player. If NULL, the minimum pitch is not reported.
	pMaxPitch	[out]	Pointer to a location to receive the maximum pitch shift supported for the player. If NULL, the maximum pitch is not reported.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The values returned by this method are the absolute minimum and maximum values supported by the implementation for the player when no other changes, such as Doppler-related changes , are applied.		
See also	None.		

8.37 SLPlayItf

Description

`SLPlayItf` is an interface for controlling the playback state of an object. The playback state machine is as follows:

Table 11: Play Head Position in Different Play States

Play State	Head forced to beginning	Prefetching	Head trying to move
Stopped	X		
Paused		X	
Playing		X	X

This interface is supported on the Audio Player [see section 7.2] and MIDI Player [see section 7.8] objects.

See Appendix B: and Appendix C: for examples using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_PLAY;

struct SLPlayItf_;
typedef const struct SLPlayItf_ * const * SLPlayItf;

struct SLPlayItf_ {
   SLresult (*SetPlayState) (
         SLPlayItf self,
         SLuint32 state
   );
   SLresult (*GetPlayState) (
         SLPlayItf self,
         SLuint32 *pState
   );
   SLresult (*GetDuration) (
         SLPlayItf self,
         SLmillisecond *pMsec
   );
   SLresult (*GetPosition) (
         SLPlayItf self,
         SLmillisecond *pMsec
   );
   SLresult (*RegisterCallback) (
         SLPlayItf self,
         slPlayCallback callback,
         void *pContext
   );
   SLresult (*SetCallbackEventsMask) (
         SLPlayItf self,
         SLuint32 eventFlags
   );
   SLresult (*GetCallbackEventsMask) (
         SLPlayItf self,
         SLuint32 *pEventFlags
   );
   SLresult (*SetMarkerPosition) (
         SLPlayItf self,
         SLmillisecond mSec
   );
   SLresult (*ClearMarkerPosition) (
         SLPlayItf self
   );
```

```
    SLresult (*GetMarkerPosition) (
          SLPlayItf self,
          SLmillisecond *pMsec
    );
    SLresult (*SetPositionUpdatePeriod) (
          SLPlayItf self,
          SLmillisecond mSec
    );
    SLresult (*GetPositionUpdatePeriod) (
          SLPlayItf self,
          SLmillisecond *pMsec
    );
};
```

Interface ID

ef0bd9c0-ddd7-11db-bf49-0002a5d5c51b

Defaults

Initially, the playback state is SL_PLAYSTATE_STOPPED, the position is at the beginning of the content, the update period is one second, and there are no markers set nor callbacks registered and the callback event flags are cleared.

Callbacks

slPlayCallback

```
typedef void (SLAPIENTRY *slPlayCallback) (
    SLPlayItf caller,
    void *pContext,
    SLuint32 event
);
```

Description	Notifies the player application of a playback event.		
Parameters	caller	[in]	Interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	event	[in]	Indicates a bit-mask of one or more events that have occurred (see SL_PLAYEVENT macros in section 9.2.35).
Comments	None.		
See also	RegisterCallback()		

Methods

SetPlayState			
`SLresult (*SetPlayState) ( SLPlayItf self, SLuint32 state );`			
Description	Requests a transition of the player into the given play state.		
Pre-conditions	None. The player may be in any state.		
Parameters	self	[in]	Interface self-reference.
	state	[in]	Desired playback state. Must be one of the SL_PLAYSTATE defines [see section 9.2.36].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_PERMISSION_DENIED SL_RESULT_CONTENT_CORRUPTED SL_RESULT_CONTENT_UNSUPPORTED		
Comments	All state transitions are legal. The state defaults to SL_PLAYSTATE_STOPPED. Note that although the state change is immediate, there may be some latency between the execution of this method and its effect on behavior. In this sense, a player's state technically represents the application's intentions for the player. Note that the player's state has an effect on the player's prefetch status (see SLPrefetchStatusItf for details). The player may return SL_RESULT_PERMISSION_DENIED, SL_RESULT_CONTENT_CORRUPTED or SL_RESULT_CONTENT_UNSUPPORTED respectively if, at the time a state change is requested, it detects insufficient permissions, corrupted content, or unsupported content. When the player reaches the end of content, the play state will transition to paused and the play cursor will remain at the end of the content.		

GetPlayState			
`SLresult (*GetPlayState) ( SLPlayItf self, SLuint32 *pState );`			
Description	Gets the player's current play state.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pState	[out]	Pointer to a location to receive the current play state of the player. The state returned is one of the SL_PLAYSTATE macros [see section 9.2.36]. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

GetDuration			
`SLresult (*GetDuration) ( SLPlayItf self, SLmillisecond *pMsec );`			
Description	Gets the duration of the current content, in milliseconds.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMsec	[out]	Pointer to a location to receive the number of milliseconds corresponding to the total duration of this current content. If the duration is unknown, this value shall be SL_TIME_UNKNOWN [see section 9.2.51]. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	In the case where the data source is a buffer queue, the current duration is the cumulative duration of all buffers since the last buffer queue Clear() method.		

GetPosition

```
SLresult (*GetPosition) (
    SLPlayItf self,
    SLmillisecond *pMsec
);
```

Description	Returns the current position of the playback head relative to the beginning of the content.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMsec	[out]	Pointer to a location to receive the position of the playback head relative to the beginning of the content, and is expressed in milliseconds. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The returned value is bounded between 0 and the duration of the content. In the case where the data source is a buffer queue, the current position is the cumulative duration of all buffers since the last buffer queue Clear() method. Note that the position is defined relative to the content playing at 1x forward rate; positions do not scale with changes in playback rate.		

RegisterCallback

```
SLresult (*RegisterCallback) (
   SLPlayItf self,
   slPlayCallback callback,
   void *pContext
);
```

Description	Sets the playback callback function.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`callback`	[in]	Callback function invoked when one of the specified events occurs. A `NULL` value indicates that there is no callback.
	`pContext`	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The callback function defaults to `NULL`. The context pointer can be used by the application to pass state to the callback function.		

SetCallbackEventsMask

```
SLresult (*SetCallbackEventsMask) (
    SLPlayItf self,
    SLuint32 eventFlags
);
```

Description	Enables/disables notification of playback events.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	eventFlags	[in]	Bitmask of play event flags (see SL_PLAYEVENT macros in section 9.2.35) indicating which callback events are enabled. The presence of a flag enables notification for the corresponding event. The absence of a flag disables notification for the corresponding event.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The callback event flags default to all flags cleared.		

GetCallbackEventsMask

```
SLresult (*GetCallbackEventsMask) (
    SLPlayItf self,
    SLuint32 *pEventFlags
);
```

Description	Queries for the notification state (enabled/disabled) of playback events.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pEventFlags	[out]	Pointer to a location to receive the bitmask of play event flags (see SL_PLAYEVENT macros in section 9.2.35) indicating which callback events are enabled. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetMarkerPosition

```
SLresult (*SetMarkerPosition) (
   SLPlayItf self,
   SLmillisecond mSec
);
```

Description	Sets the position of the playback marker.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	mSec	[in]	Position of the marker expressed in milliseconds and relative to the beginning of the content.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The SL_PLAYEVENT_HEADATMARKER [see section 9.2.35] event will be triggered when any of the following happens: • The player object is in the play state and the playback head goes from less than or equal to greater than or equal to the marker position. • The player object is in the play state and both the playback head and marker are 0. By default, there is no marker position defined. When a marker position coincides with a periodic position update (as specified by SetPositionUpdatePeriod()), then both the marker position callback and the periodic position update callback must be posted next to each other. The order of the two callbacks is insignificant.		
See Also	ClearMarkerPosition(), SetPositionUpdatePeriod()		

ClearMarkerPosition

```
SLresult (*ClearMarkerPosition) (
    SLPlayItf self
);
```

Description	Clears marker.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This function succeeds even if the marker is already clear.		
See Also	SetMarkerPosition()		

GetMarkerPosition

```
SLresult (*GetMarkerPosition) (
   SLPlayItf self,
   SLmillisecond *pMsec
);
```

Description	Queries the position of playback marker.		
Pre-conditions	A marker has been set (using SetMarkerPosition())		
Parameters	self	[in]	Interface self-reference.
	pMsec	[out]	Pointer to a location to receive the position of the marker expressed in milliseconds, relative to the beginning of the content. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	None.		
See Also	SetMarkerPosition(), ClearMarkerPosition()		

SetPositionUpdatePeriod

```
SLresult (*SetPositionUpdatePeriod) (
   SLPlayItf self,
   SLmillisecond mSec
);
```

Description	Sets the interval between periodic position notifications.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`mSec`	[in]	Non-zero period between position notifications in milliseconds.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	The player will notify the application when the playback head passes through the positions implied by the specified period. Those positions are defined as the whole multiples of the period relative to the beginning of the content. By default, the update period is 1000 milliseconds. When a periodic position update coincides with a marker position(as specified by `SetMarkerPosition()`), then both the position update period callback and the marker position callback must be posted next to each other. The order of the two callbacks is insignificant.		
See Also	`SetMarkerPosition()`		

GetPositionUpdatePeriod

```
SLresult (*GetPositionUpdatePeriod) (
   SLPlayItf self,
   SLmillisecond *pMsec
);
```

Description	Queries the interval between periodic position notifications.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pMsec`	[out]	Pointer to a location to receive the period between position notifications in milliseconds. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

8.38 SLPlaybackRateItf

Description

`SLPlaybackRateItf` is an interface for controlling setting and retrieving the rate at which an object presents data. Rates are expressed as a permille type (namely, parts per thousand):

- Negative values indicate reverse presentation.
- A value of 0‰ indicates paused presentation.
- Positive values less than 1000‰ indicate slow forward rates.
- A value of 1000‰ indicates normal 1X forward playback.
- Positive values greater than 1000‰ indicate fast forward rates.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_PLAYBACKRATE;

struct SLPlaybackRateItf_;
typedef const struct SLPlaybackRateItf_ * const * SLPlaybackRateItf;

struct SLPlaybackRateItf_ {
   SLresult (*SetRate)(
         SLPlaybackRateItf self,
         SLpermille rate
   );
   SLresult (*GetRate)(
         SLPlaybackRateItf self,
         SLpermille *pRate
   );
   SLresult (*SetPropertyConstraints)(
         SLPlaybackRateItf self,
         SLuint32 constraints
   );
   SLresult (*GetProperties)(
         SLPlaybackRateItf self,
         SLuint32 *pProperties
   );
   SLresult (*GetCapabilitiesOfRate)(
         SLPlaybackRateItf self,
         SLpermille rate,
         SLuint32 *pCapabilities
   );
   SLresult (*GetRateRange) (
         SLPlaybackRateItf self,
         SLuint8 index,
         SLpermille *pMinRate,
         SLpermille *pMaxRate,
         SLpermille *pStepSize,
         SLuint32 *pCapabilities
   );
};
```

Interface ID

2e3b2a40-ddda-11db-a349-0002a5d5c51b

Defaults

The rate value defaults to 1000‰ (that is, normal 1X forward playback).

Methods

SetRate

```
SLresult (*SetRate) (
    SLPlaybackRateItf self,
    SLpermille rate
);
```

Description	Sets the rate of presentation.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	rate	[in]	Desired rate.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	1000 is the default rate. The application may query supported rates via the GetRateRange() method. The SL_RESULT_FEATURE_UNSUPPORTED return value accommodates the circumstance where the content being played does not afford adjustments of the playback rate.		

GetRate

```
SLresult (*GetRate) (
    SLPlaybackRateItf self,
    SLpermille *pRate
);
```

Description	Gets the rate of presentation.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pRate	[out]	Pointer to a location to receive the rate of the player. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetPropertyConstraints			
`SLresult (*SetPropertyConstraints) ( SLPlaybackRateItf self, SLuint32 constraints );`			
Description	Sets the current rate property constraints.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`constraints`	[in]	Bitmask of the allowed rate properties requested (see SL_RATEPROP macros in section 9.2.42). An implementation may choose any of the given properties to implement rate and none of the excluded properties. If the bitmask is not well-formed, this method returns SL_RESULT_PARAMETER_INVALID. If the constraints cannot be satisfied, this method returns SL_RESULT_FEATURE_UNSUPPORTED.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Note that rate property capabilities may vary from one rate to another. This implies that a setting supported for one rate may be unsupported for another. The default audio properties are SL_RATEPROP_NOPITCHCORAUDIO.		

GetProperties			
`SLresult (*GetProperties) ( SLPlaybackRateItf self, SLuint32 *pProperties );`			
Description	Gets the current properties.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pProperties	[out]	Pointer to a location to receive the bitmask expressing the current rate properties. The range of the bitmask is defined by the SL_RATEPROP macros [see section 9.2.42]. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

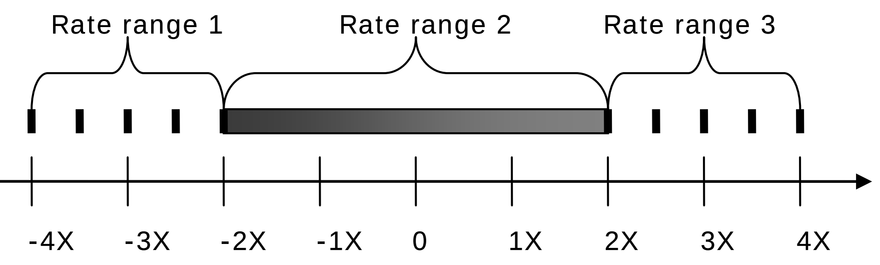
GetCapabilitiesOfRate

```
SLresult (*GetCapabilitiesOfRate) (
   SLPlaybackRateItf self,
   SLpermille rate,
   SLuint32 *pCapabilities
);
```

Description	Gets the capabilities of the specified rate.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	rate	[in]	Rate for which the capabilities are being queried.
	pCapabilities	[out]	Pointer to a location to receive the bitmask expressing capabilities of the given rate in terms of rate properties (see SL_RATEPROP macros in section 9.2.42).
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	A application may also leverage this method to verify that a particular rate is supported.		

GetRateRange			
`SLresult (*GetRateRange) ( SLPlaybackRateItf self, SLuint8 index, SLpermille *pMinRate, SLpermille *pMaxRate, SLpermille *pStepSize, SLuint32 *pCapabilities );`			
Description	Retrieves the ranges of rates supported.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	index	[in]	Index of the range being queried. If an implementation supports n rate ranges, this value is between 0 and (n-1) and all values greater than n cause the method to return SL_RESULT_PARAMETER_INVALID.
	pMinRate	[out]	Pointer to a location to receive the minimum rate supported. May be negative or positive. Must be equal to or less than maxRate. This must be non-NULL.
	pMaxRate	[out]	Pointer to a location to receive the maximum rate supported. May be negative or positive. Must be equal to or greater than minRate. This must be non-NULL.
	pStepSize	[out]	Pointer to a location to receive the distance between one rate and an adjacent rate in the range. A value of zero denotes a continuous range. This must be non-NULL.
	pCapabilities	[out]	Pointer to a location to receive the bitmask of supported rate properties in the given range. The range of the bitmask is defined by the SL_RATEPROP macros [see section 9.2.42]. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	An implementation expresses the set of supported rates as one or more ranges. Each range is defined by the lowest and highest rates in the range, the step size between these bounds, and the rate properties of this range. If all rates an implementation supports are evenly spaced and have		

	same capabilities, `GetRateRange()` method may return a single range. If not, the `GetRateRange()` method will return as many ranges as necessary in order to adequately express the set of rates (and associated properties) supported. In this case, the application must call `GetRateRange()` multiple times to query all the ranges; `GetRateRange()` returns only one range per call.



Rate range examples: Range 1 has a min of -4000‰, a max of -2000‰ and a step of 500‰. Range 2 has a min of -2000‰, a max of 2000‰, and a step of 0‰. Range 3 has a min of 2000‰, a max of 4000‰ and a step of 500‰.

Figure 38: Example rate ranges

8.39 SLPrefetchStatusItf

Description

`SLPrefetchStatusItf` is an interface for querying the prefetch status of a player.

The prefetch status is a continuum ranging from no data prefetched to the maximum amount of data prefetched. It includes a range where underflow may occur and a range where there is a sufficient amount of data present. The underflow and sufficient data ranges may not relate to fixed fill level positions, but be implementation dependent and dynamically vary based on factors as e.g. buffering length, consumption rate, communication latency, hysteresis, etc. The prefetch status interface allows an application to query for prefetch status or register prefetch status callbacks. The latency of status and fill level callbacks are implementation dependent.

One example usage of the `SLPrefetchStatusItf` is to order the player into paused state when receiving an underflow event and into play state when receiving a sufficient data event when playing network stored media sources. Another example usage is to display fill level percentage to the end user by using the callback and the `GetFillLevel` method.

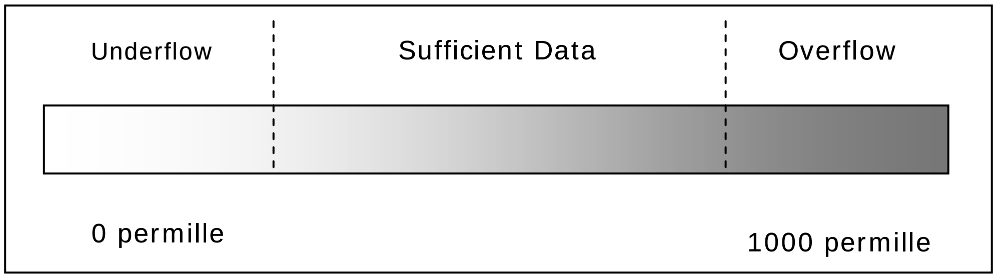


Figure 39: Prefetch continuum range

This interface is supported on Audio Player [see section 7.2] and MIDI Player [see section 7.8] objects.

See section C.4 for an example using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_PREFETCHSTATUS;

struct SLPrefetchStatusItf_;
typedef const struct SLPrefetchStatusItf_
   * const * SLPrefetchStatusItf;

struct SLPrefetchStatusItf_ {
   SLresult (*GetPrefetchStatus) (
         SLPrefetchStatusItf self,
         SLuint32 *pStatus
   );
   SLresult (*GetFillLevel) (
         SLPrefetchStatusItf self,
         SLpermille *pLevel
   );
   SLresult (*RegisterCallback) (
         SLPrefetchStatusItf self,
         slPrefetchCallback callback,
         void *pContext
   );
   SLresult (*SetCallbackEventsMask) (
         SLPrefetchStatusItf self,
         SLuint32 eventFlags
   );
   SLresult (*GetCallbackEventsMask) (
         SLPrefetchStatusItf self,
         SLuint32 *pEventFlags
   );
   SLresult (*SetFillUpdatePeriod) (
         SLPrefetchStatusItf self,
         SLpermille period
   );
   SLresult (*GetFillUpdatePeriod) (
         SLPrefetchStatusItf self,
         SLpermille *pPeriod
   );
   SLresult (*GetError) (
         SLPrefetchStatusItf self,
         SLresult *pResult
   );
}
```

Interface ID

275229d0-c5db-11df-bd3b-0800200c9a66

Defaults

Initially, there is no callback registered, the fill update period is 100 permille, and the event flags are clear.

Callbacks

slPrefetchCallback

```
typedef void (SLAPIENTRY *slPrefetchCallback) (
    SLPrefetchStatusItf caller,
    void *pContext,
    SLuint32 event
);
```

Description	Notifies the player application of a prefetch event.		
Parameters	caller	[in]	Interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	event	[in]	Event that has occurred (see SL_PREFETCHEVENT macros in section 9.2.37).
Comments	None.		
See also	RegisterCallback()		

Methods

GetPrefetchStatus			
`SLresult (*GetPrefetchStatus) ( SLPrefetchStatusItf self, SLuint32 *pStatus );`			
Description	Gets the player's current prefetch status.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pStatus`	[out]	Pointer to a location to receive the current prefetch status of the player. The status returned is one of the SL_PREFETCHSTATUS defines [see section 9.2.38]. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		

GetFillLevel			
`SLresult (*GetFillLevel) ( SLPrefetchStatusItf self, SLpermille *pLevel );`			
Description	Queries the fill level of the prefetch.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pLevel`	[out]	Pointer to a location to receive the data fill level in parts per thousand. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The fill level is not tied to specific buffer within a player, but indicates more abstractly the progress a player has made in preparing data for playback. This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		

RegisterCallback

```
SLresult (*RegisterCallback) (
   SLPrefetchStatusItf self,
   slPrefetchCallback callback,
   void *pContext
);
```

Description	Sets the prefetch callback function.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	callback	[in]	Callback function invoked when one of the specified events occurs. A NULL value indicates that there is no callback.
	pContext	[in]	User context data that is to be returned as part of the callback method.
Return value	The return value can be the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Callback function defaults to NULL. The context pointer can be used by the application to pass state to the callback function. This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		
See Also	SL_PREFETCHEVENT macros [see section 9.2.37].		

SetCallbackEventsMask			
`SLresult (*SetCallbackEventsMask) ( SLPrefetchStatusItf self, SLuint32 eventFlags );`			
Description	Sets the notification state of the prefetch events.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	eventFlags	[in]	Bitmask of prefetch event flags (see SL_PREFETCHEVENT macros in section 9.2.37) indicating which callback events are enabled.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Event flags default to all flags cleared. This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		

GetCallbackEventsMask			
`SLresult (*GetCallbackEventsMask) ( SLPrefetchStatusItf self, SLuint32 *pEventFlags );`			
Description	Queries the notification state of the prefetch events.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pEventFlags	[out]	Pointer to a location to receive the bitmask of prefetch event flags (see SL_PREFETCHEVENT macros in section 9.2.37) indicating which callback events are enabled. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_FEATURE_UNSUPORTED		
Comments	This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		

SetFillUpdatePeriod

```
SLresult (*SetFillUpdatePeriod) (
    SLPrefetchStatusItf self,
    SLpermille period
);
```

Description	Sets the notification period for fill level updates. This period implies the set discrete fill level values that will generate notifications from the player.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`period`	[in]	Non-zero period between fill level notifications in permille units.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Non-zero period between fill level notifications in permille. Notifications will occur at 0 permille (i.e. empty) and at whole number increments of the period from 0. For instance, if the period is 200 permille (i.e. 20%), then the player will generate a notification when 0%, 20%, 40%, 60%, 80%, or 100% full. The default period is 100 permille. This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		

GetFillUpdatePeriod

```
SLresult (*GetFillUpdatePeriod) (
    SLPrefetchStatusItf self,
    SLpermille *pPeriod
);
```

Description	Queries the notification period for fill level updates.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pPeriod	[out]	Pointer to a location to receive the period between fill level notifications in permille units. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		

GetError

```
SLresult (*GetError) (
    SLPrefetchStatusItf self,
    SLresult *pResult
);
```

Description	Retrieves the last error code that was encountered since prefetching was started on this player.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pResult`	[out]	Pointer to a location to receive the error code.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The registered `slPrefetchCallback` will be called once for every error encountered during the prefetch operation. During the execution of the `slPrefetchCallback`, the application can call `GetError()` to retrieve the error code for the error encountered. Failure to retrieve the error code during the execution of the callback results in undefined behavior. If no error is available for retrieval, SL_RESULT_SUCCESS will be returned. The following error codes may be returned: SL_RESULT_MEMORY_FAILURE SL_RESULT_RESOURCE_ERROR SL_RESULT_RESOURCE_LOST SL_RESULT_IO_ERROR SL_RESULT_CONTENT_CORRUPTED SL_RESULT_CONTENT_UNSUPPORTED SL_RESULT_CONTENT_NOT_FOUND SL_RESULT_PERMISSION_DENIED SL_RESULT_INTERNAL_ERROR SL_RESULT_UNKNOWN_ERROR SL_RESULT_OPERATION_ABORTED This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		

8.40 SLPresetReverbItf

Description

This interface allows an application to configure the global reverb using a reverb preset. This is primarily used for adding some reverb in a music playback context. Applications requiring control over a more advanced environmental reverb are advised to use the `SLEnvironmentalReverbItf` interface, where available.

When this interface is exposed on the Output Mix, it acts as an auxiliary effect; for reverb to be applied to a player's output, the `SLEffectSendItf` interface [see section 8.17] must be exposed on the player.

The following restriction must be adhered to when exposing this interface:

- It is not possible to expose this interface while the `SLEnvironmentalReverbItf` interface of the same object is already exposed.

This interface is supported on Output Mix [see section 7.9] objects.

See section C.2 for an example using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_PRESETREVERB;

struct SLPresetReverbItf_;
typedef const struct SLPresetReverbItf_ * const * SLPresetReverbItf;

struct SLPresetReverbItf_ {
   SLresult (*SetPreset) (
         SLPresetReverbItf self,
         SLuint16 preset
   );
   SLresult (*GetPreset) (
         SLPresetReverbItf self,
         SLuint16 *pPreset
   );
};
```

Interface ID

47382d60-ddd8-11db-bf3a-0002a5d5c51b

Defaults

Reverb preset: `SL_REVERBPRESET_NONE`.

Methods

SetPreset			
`SLresult (*SetPreset) ( SLPresetReverbItf self, SLuint16 preset );`			
Description	Enables a preset on the global reverb.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	preset	[in]	Reverb preset. This must be one of the SL_REVERBPRESET presets listed [see section 9.2.45].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	The reverb SL_REVERBPRESET_NONE disables any reverb from the current output but does not free the resources associated with the reverb. For an application to signal to the implementation to free the resources, it must either change the object's state to Unrealized or dynamically remove the SLReverbPresetItf interface using SLDynamicInterfaceManagementItf::RemoveInterface [see section 8.17]. Some implementations may support an extended set of reverb presets, in which case a wider range of reverb presets are accepted in this method.		

GetPreset			
`SLresult (*GetPreset) ( SLPresetReverbItf self, SLuint16 *pPreset );`			
Description	Gets the current global reverb preset.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pPreset`	[out]	Pointer to location for the current reverb preset. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

8.41 SLRatePitchItf

Description

SLRatePitchItf is an interface for controlling the rate at which a sound is played back. A change in rate will cause a change in pitch. The rate is specified using a permille factor that controls the playback rate:

- A value of 1000 ‰ indicates normal playback rate; there is no change in pitch.
- A value of 500 ‰ indicates playback at half the normal rate, causing a pitch shift of –12 semitones (one octave decrease).
- A value of 2000 ‰ indicates playback at double the normal rate, causing a pitch shift of 12 semitones (one octave increase).

This interface is supported on the Audio Player [see section 7.2] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_RATEPITCH;

struct SLRatePitchItf_;
typedef const struct SLRatePitchItf_ * const * SLRatePitchItf;

struct SLRatePitchItf_ {
   SLresult (*SetRate) (
         SLRatePitchItf self,
         SLpermille rate
   );
   SLresult (*GetRate) (
         SLRatePitchItf self,
         SLpermille *pRate
   );
   SLresult (*GetRatePitchCapabilities) (
         SLRatePitchItf self,
         SLpermille *pMinRate,
         SLpermille *pMaxRate
   );
};
```

Interface ID

61b62e60-ddda-11db-9eb8-0002a5d5c51b

Defaults

Rate: 1000 ‰

Methods

SetRate

```
SLresult (*SetRate) (
    SLRatePitchItf self,
    SLpermille rate
);
```

Description	Sets a player's rate.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`rate`	[in]	Rate factor in permille. The range supported by this parameter is both implementation- and content-dependent and can be determined using the `GetRatePitchCapabilities()` method.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	Changing a player's rate will change the pitch at which a sound is played. For example, when the rate is set at 2000 ‰, the sound will be played at twice the normal playback rate, causing a one octave increase in pitch. As the Doppler effect may be implemented internally as a rate change and the rate change may be capped, in some situations, such as when the rate change exceeds the device capabilities, this method may appear to have no audible effect, even though the method succeeded. In this case, the effect will be audible when the amount of Doppler is reduced.		
See also	`SL3DDopplerItf` [see section 8.3].		

GetRate

```
SLresult (*GetRate) (
   SLRatePitchItf self,
   SLpermille *pRate,
);
```

Description	Gets the player’s current rate.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pRate	[out]	Pointer to location for the player’s rate in permille. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	SL3DDopplerItf [see section 8.3].		

GetRatePitchCapabilities

```
SLresult (*GetRatePitchCapabilities) (
    SLRatePitchItf self,
    SLpermille *pMinRate,
    SLpermille *pMaxRate
);
```

Description	Retrieves the player's rate pitch capabilities.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMinRate	[out]	Pointer to a location for minimum rate supported for the player. If NULL, the minimum rate is not reported.
	pMaxRate	[out]	Pointer to a location for the maximum rate supported for the player. If NULL, the maximum rate is not reported.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS		
Comments	The values returned by this method are the absolute minimum and maximum values supported by the implementation for the player when no other Doppler or rate changes are applied.		
See also	None.		

8.42 SLRecordItf

Description

`SLRecordItf` is an interface for controlling the recording state of an object. The record state machine is as follows:

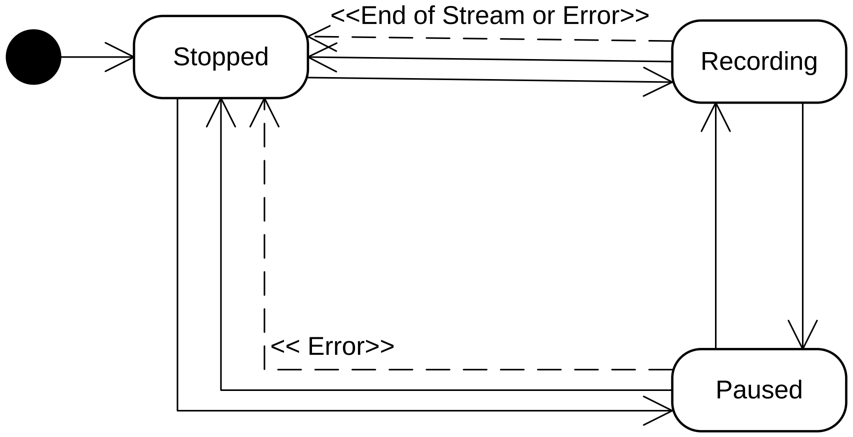


Figure 40: Record state machine

Table 12: Data Status and Recording State

Recording State	Destination[1] closed	Head[2] moving (sending data to destination)
Stopped	X	
Paused[3]		
Recording		X

In case the storage media become full while recording to a file, SL_OBJECT_EVENT_RUNTIME_ERROR will be posted via `slObjectCallback` with SL_RESULT_IO_ERROR as this callback's result parameter. The recorder will in that case autotransition into SL_RECORDSTATE_STOPPED state.

[1] "Destination" denotes the sink of the recording process (for example, a file being written to).

[2] "Head" denotes the position of the recording process relative in time to the duration of the entire recording (for example, if the five seconds of audio have been sent to the destination, the head is at five seconds).

[3] If a recorder transitions from Paused to Recording (without an intervening transition

to Stopped), the newly captured data is appended to data already sent to the destination.

This interface is supported on the Audio Recorder [see section 7.2] object.

See section B.1.2 for an example using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_RECORD;

struct SLRecordItf_;
typedef const struct SLRecordItf_ * const * SLRecordItf;

struct SLRecordItf_ {
   SLresult (*SetRecordState) (
         SLRecordItf self,
         SLuint32 state
   );
   SLresult (*GetRecordState) (
         SLRecordItf self,
         SLuint32 *pState
   );
   SLresult (*SetDurationLimit) (
         SLRecordItf self,
         SLmillisecond msec
   );
   SLresult (*GetPosition) (
         SLRecordItf self,
         SLmillisecond *pMsec
   );
   SLresult (*RegisterCallback) (
         SLRecordItf self,
         slRecordCallback callback,
         void *pContext
   );
   SLresult (*SetCallbackEventsMask) (
         SLRecordItf self,
         SLuint32 eventFlags
   );
   SLresult (*GetCallbackEventsMask) (
         SLRecordItf self,
         SLuint32 *pEventFlags
   );
   SLresult (*SetMarkerPosition) (
         SLRecordItf self,
         SLmillisecond mSec
   );
   SLresult (*ClearMarkerPosition) (
         SLRecordItf self
   );
```

```
      SLresult (*GetMarkerPosition) (
            SLRecordItf self,
            SLmillisecond *pMsec
      );
      SLresult (*SetPositionUpdatePeriod) (
            SLRecordItf self,
            SLmillisecond mSec
      );
      SLresult (*GetPositionUpdatePeriod) (
            SLRecordItf self,
            SLmillisecond *pMsec
      );
   };
```

Interface ID

c5657aa0-dddb-11db-82f7-0002a5d5c51b

Defaults

A recorder defaults to the SL_RECORDSTATE_STOPPED state, with no marker, no duration limit, an update period of one second, and there are no markers set nor callbacks registered and the callback event flags are cleared.

Callbacks

slRecordCallback			
`typedef void (SLAPIENTRY *slRecordCallback) ( SLRecordItf caller, void *pContext, SLuint32 event );`			
Description	Notifies the recorder application of a recording event.		
Parameters	caller	[in]	Interface instantiation on which the callback was registered.
	pContext	[in]	User context data that is supplied when the callback method is registered.
	event	[in]	Event that has occurred (see SL_RECORDEVENT macros in section 9.2.43).
Comments	None.		
See Also	RegisterCallback()		

Methods

SetRecordState			
`SLresult (*SetRecordState) ( SLRecordItf self, SLuint32 state );`			
Description	Transitions recorder into the given record state.		
Pre-conditions	None. The recorder may be in any state.		
Parameters	self	[in]	Interface self-reference.
	state	[in]	Desired recorder state. Must be one of SL_RECORDSTATE macros [see section 9.2.44].
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	All state transitions are legal.		

GetRecordState			
`SLresult (*GetRecordState) ( SLRecordItf self, SLuint32 *pState );`			
Description	Gets the recorder's current record state.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pState	[out]	Pointer to a location to receive the current record state of the recorder. See SL_RECORDSTATE macros in section 9.2.44. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetDurationLimit

```
SLresult (*SetDurationLimit) (
    SLRecordItf self,
    SLmillisecond msec
);
```

Description	Sets the duration of current content in milliseconds.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	msec	[in]	Non-zero limit on the duration of total recorded content in milliseconds. A value of SL_TIME_UNKNOWN indicates no limit.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	When the recorder reaches the limit, it automatically transitions to the SL_RECORDSTATE_STOPPED state and notifies the application via the SL_RECORDEVENT_HEADATLIMIT event [see section 9.2.43].		

GetPosition

```
SLresult (*GetPosition) (
    SLRecordItf self,
    SLmillisecond *pMsec
);
```

Description	Returns the current position of the recording head relative to the beginning of content.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMsec	[out]	Pointer to a location to receive the position of the recording head relative to the beginning of the content, expressed in milliseconds. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The position is synonymous with the amount of recorded content.		

RegisterCallback

```
SLresult (*RegisterCallback) (
   SLRecordItf self,
   slRecordCallback callback,
   void *pContext
);
```

Description	Registers the record callback function.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`callback`	[in]	Callback function invoked when one of the specified events occurs. A `NULL` value indicates that there is no callback.
	`pContext`	[in]	User context data that is to be returned as part of the callback method.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		
See Also	`SL_RECORDEVENT` [see section 9.2.43].		

SetCallbackEventsMask

```
SLresult (*SetCallbackEventsMask) (
   SLRecordItf self,
   SLuint32 eventFlags
);
```

Description	Sets the notification state of record events.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	eventFlags	[in]	Combination record event flags (see SL_RECORDEVENT macros in section 9.2.43) indicating which callback events are enabled.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The callback event flags default to all flags cleared.		

GetCallbackEventsMask

```
SLresult (*GetCallbackEventsMask) (
   SLRecordItf self,
   SLuint32 *pEventFlags
);
```

Description	Queries the notification state of record events.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pEventFlags	[out]	Pointer to a location to receive the combination of record event flags (see SL_RECORDEVENT macros in section 9.2.43) indicating which callback events are enabled. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetMarkerPosition

```
SLresult (*SetMarkerPosition) (
    SLRecordItf self,
    SLmillisecond mSec
);
```

Description	Sets the position of the recording marker.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	mSec	[in]	Position of the marker expressed in milliseconds and relative to the beginning of the content. Must be between 0 and the specified duration limit.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The SL_RECORDEVENT_HEADATMARKER [see section 9.2.43] event will be triggered when any of the following happens: • The recorder object is in the record state and the recording head goes from less than or equal to greater than or equal to the marker position. • The recorder object is in the record state and both the recording head and marker position are 0.		

ClearMarkerPosition

```
SLresult (*ClearMarkerPosition) (
    SLRecordItf self
);
```

Description	Clears marker.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This function succeeds even if the marker is already clear.		
See Also	SetMarkerPosition()		

GetMarkerPosition

```
SLresult (*GetMarkerPosition) (
    SLRecordItf self,
    SLmillisecond *pMSec
);
```

Description	Queries the position of the recording marker.		
Pre-conditions	A marker has been set (using `SetMarkerPosition()`)		
Parameters	`self`	[in]	Interface self-reference.
	`pMSec`	[out]	Pointer to a location to receive the position of the marker expressed in milliseconds and relative to the beginning of the content. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		

SetPositionUpdatePeriod

```
SLresult (*SetPositionUpdatePeriod) (
    SLRecordItf self,
    SLmillisecond mSec
);
```

Description	Sets the interval between periodic position notifications.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`mSec`	[in]	Non-zero period between position notifications in milliseconds.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	The recorder will notify the application when the recording head passes through the positions implied by the specified period. Those positions are defined as the whole multiples of the period relative to the beginning of the content.		

GetPositionUpdatePeriod			
`SLresult (*GetPositionUpdatePeriod) ( SLRecordItf self, SLmillisecond *pMSec );`			
Description	Queries the interval between periodic position notifications.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMSec	[out]	Pointer to a location to receive the period between position notifications in milliseconds. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.43 SLSeekItf

Description

SeekItf is an interface for manipulating a playback head, including setting its position and looping characteristics. When supported, seeking may be used, regardless of playback state or rate.

This interface is supported on the Audio Player [see section 7.2] and MIDI Player [see section 7.8] objects.

See sections B.5.2, B.6.1, and Appendix C: for examples using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_SEEK;

struct SLSeekItf_;
typedef const struct SLSeekItf_ * const * SLSeekItf;

struct SLSeekItf_ {
   SLresult (*SetPosition)(
         SLSeekItf self,
         SLmillisecond pos,
         SLuint32 seekMode
   );
   SLresult (*SetLoop)(
         SLSeekItf self,
         SLboolean loopEnable,
         SLmillisecond startPos,
         SLmillisecond endPos
   );
   SLresult (*GetLoop)(
         SLSeekItf self,
         SLboolean *pLoopEnabled,
         SLmillisecond *pStartPos,
         SLmillisecond *pEndPos
   );
};
```

Interface ID

d43135a0-dddc-11db-b458-0002a5d5c51b

Defaults

The playback position defaults to 0 milliseconds (the beginning of the current content). Global and local looping are disabled by default.

Methods

SetPosition			
`SLresult (*SetPosition) ( SLSeekItf self, SLmillisecond pos, SLuint32 seekMode );`			
Description	Sets the position of the playback head.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pos`	[in]	Desired playback position in milliseconds, relative to the beginning of content.
	`seekMode`	[in]	Inherent seek mode. See the seek mode definition for details [see section 9.2.49]. If the seek mode is not supported, this method will return `SL_RESULT_FEATURE_UNSUPPORTED`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	The implementation may set the position to the nearest discrete sample or frame. Note that the position is defined relative to the content playing at 1x forward rate; positions do not scale with changes in playback rate. This method will return `SL_RESULT_FEATURE_UNSUPPORTED` if the media object data source is a buffer queue.		

SetLoop

```
SLresult (*SetLoop) (
    SLSeekItf self,
    SLboolean loopEnable,
    SLmillisecond startPos,
    SLmillisecond endPos
);
```

Description	Enables or disables looping and sets the start and end points of looping. When looping is enabled and the playback head reaches the end position, the player automatically sets the head to the start position and remains in the SL_PLAYSTATE_PLAYING state. Setting a loop does not otherwise have any effect on the playback head even if the head is outside the loop at the time the loop is set. The start and end positions are still set when looping is disabled.		
Pre-conditions	Specified end position is greater than specified start position.		
Parameters	self	[in]	Interface self-reference.
	loopEnable	[in]	Specifies whether looping is enabled (true) or disabled (false).
	startPos	[in]	Position in milliseconds relative to the beginning of content specifying the start of the loop.
	endPos	[in]	Position in milliseconds relative to the beginning of content specifying the end the loop. endPos must be greater than startPos. A value of SL_TIME_UNKNOWN denotes the end of the stream.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue. If local looping is not supported, this method returns SL_RESULT_FEATURE_UNSUPPORTED. **PROFILE NOTE** *In the Phone profile, audio players are mandated only to support SetLoop where startPos = 0, endPos = SL_TIME_UNKNOWN.* *For all the others profiles and objects SetLoop is mandated as defined in the object's profile notes.*		

GetLoop

```
SLresult (*GetLoop) (
   SLSeekItf self,
   SLboolean *pLoopEnabled,
   SLmillisecond *pStartPos,
   SLmillisecond *pEndPos
);
```

Description	Queries whether looping is enabled or disabled, and retrieves loop points.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pLoopEnabled	[out]	Pointer to a location to receive the flag indicating whether looping is enabled (true) or disabled (false). This must be non-NULL.
	pStartPos	[out]	Pointer to a location to receive the position in milliseconds relative to the beginning of content specifying the start of the loop. This must be non-NULL.
	pEndPos	[out]	Pointer to a location to receive the position in milliseconds relative to the beginning of content specifying the end the loop. A value of SL_TIME_UNKNOWN denotes the end of the stream. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	This method will return SL_RESULT_FEATURE_UNSUPPORTED if the media object data source is a buffer queue.		

8.44 SLThreadSyncItf

Description

Registered callbacks can be invoked concurrently to application threads and even concurrently to other callbacks. The application cannot assume anything about the context from which registered callbacks are invoked, and thus using the native synchronization mechanisms for synchronization of callback contexts is not guaranteed to work.

For this purpose, a critical section mechanism is introduced. There is one critical section per engine object. Applications that require more flexibility can implement such a mechanism on top of this critical section mechanism.

The semantics of the critical section mechanism are specified as follows:

- The engine is said to be **in a critical section state** during the time between when a call to `EnterCriticalSection()` has returned successfully and until the time when a call to `ExitCriticalSection()` is made.
- When the engine is in a critical section state, any call to `EnterCriticalSection()` will block until the engine exited the critical section state, or until an error has occurred (the return code of the `EnterCriticalSection()` call will reflect which of the conditions has occurred).

One important point is worth mentioning: when the engine is operating in non-thread-safe mode, the `EnterCriticalSection()` and `ExitCriticalSection()` methods are **not thread safe**, in the sense that their behavior is undefined if the application calls them from within multiple **application contexts** concurrently. These methods will, however, work properly when invoked from a single application context in concurrency with one or more **callback contexts**.

This interface is supported on the engine [see section 7.4] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_THREADSYNC;

struct SLThreadSyncItf_;
typedef const struct SLThreadSyncItf_ * const * SLThreadSyncItf;

struct SLThreadSyncItf_ {
   SLresult (*EnterCriticalSection) (
         SLThreadSyncItf self
   );
```

```
    SLresult (*ExitCriticalSection) (
        SLThreadSyncItf self
    );
};
```

Interface ID

f6ac6b40-dddc-11db-a62e-0002a5d5c51b

Defaults

Not in critical section state.

Methods

EnterCriticalSection			
`SLresult (*EnterCriticalSection) ( SLThreadSyncItf self );`			
Description	Blocks until the engine is not in critical section state, then transitions the engine into critical section state.		
Pre-conditions	The calling context must not already be in critical section state.		
Parameters	`self`	[in]	Interface self-reference.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	Use this method to achieve synchronization between application context and callback context(s), or between multiple callback contexts. See comments in the description section regarding thread-safety of this method.		
See also	`ExitCriticalSection()`		

ExitCriticalSection			

`SLresult (*ExitCriticalSection) (`
`   SLThreadSyncItf self`
`);`			

Description	Transitions the engine from critical section state to non-critical section state.		
Pre-conditions	The engine must be in critical section state. The call must be made from the same context that entered the critical section.		
Parameters	self	[in]	Interface self-reference.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PRECONDITIONS_VIOLATED		
Comments	Use this method to achieve synchronization between client application context and callback context(s), or between multiple callback contexts. See comment in description section regarding thread-safety of this method.		
See also	EnterCriticalSection()		

8.45 SLVibraItf

Description

`SLVibraItf` interface is used to activate and deactivate the Vibra I/O device object, as well as to set its frequency and intensity, if supported.

`SLVibraItf` uses the following state model, which indicates whether the vibration device is vibrating or not:

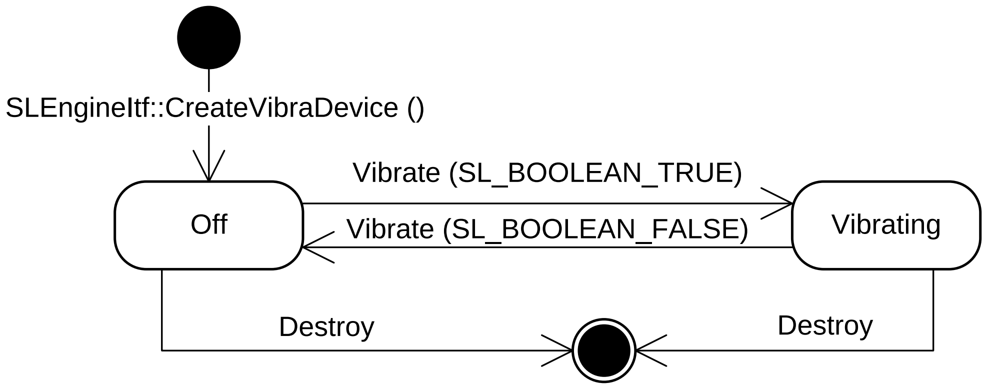

SLEngineItf::CreateVibraDevice ()
Vibrate (SL_BOOLEAN_TRUE)
Off
Vibrate (SL_BOOLEAN_FALSE)
Vibrating
Destroy
Destroy

Figure 41: Vibra I/O device state model

This interface is supported on the Vibra I/O device object [see section 7.10].

Prototype

```
SL_API extern const SLInterfaceID SL_IID_VIBRA;

struct SLVibraItf_;
typedef const struct SLVibraItf_ * const * SLVibraItf;

struct SLVibraItf_ {
   SLresult (*Vibrate) (
         SLVibraItf self,
         SLboolean vibrate
   );
   SLresult (*IsVibrating) (
         SLVibraItf self,
         SLboolean *pVibrating
   );
   SLresult (*SetFrequency) (
         SLVibraItf self,
         SLmilliHertz frequency
   );
   SLresult (*GetFrequency) (
         SLVibraItf self,
         SLmilliHertz *pFrequency
   );
   SLresult (*SetIntensity) (
         SLVibraItf self,
         SLpermille intensity
   );
   SLresult (*GetIntensity) (
         SLVibraItf self,
         SLpermille *pIntensity
   );
};
```

Interface ID

169a8d60-dddd-11db-923d-0002a5d5c51b

Defaults

Initially, the object is in the off state. Default frequency and intensity are undefined.

Methods

Vibrate

```
SLresult (*Vibrate) (
   SLVibraItf self,
   SLboolean vibrate
);
```

Description	Activates or deactivates vibration for the I/O device.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	vibrate	[in]	Boolean indicating whether to vibrate.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_IO_ERROR SL_RESULT_CONTROL_LOST		
Comments	None.		
See also	None.		

IsVibrating

```
SLresult (*IsVibrating) (
   SLVibraItf self,
   SLboolean *pVibrating
);
```

Description	Returns whether the I/O device is vibrating.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pVibrating	[out]	Address to store a boolean indicating whether the I/O device is vibrating.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

SetFrequency			
`SLresult (*SetFrequency) ( SLVibraItf self, SLmilliHertz frequency );`			
Description	Sets the vibration frequency of the I/O device.		
Pre-conditions	The Vibra I/O device must support setting frequency, per `SLVibraDescriptor::supportsFrequency`.		
Parameters	`self`	[in]	Interface self-reference.
	`frequency`	[in]	Frequency of vibration. Range is [`SLVibraDescriptor::minFrequency`, `SLVibraDescriptor::maxFrequency`]

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_RESOURCE_LOST`
	`SL_RESULT_CONTROL_LOST`		

Comments	None.		
See also	`SLVibraDescriptor` [see section 9.1.27].		

GetFrequency

```
SLresult (*GetFrequency) (
   SLVibraItf self,
   SLmilliHertz *pFrequency
);
```

Description	Returns the vibration frequency of the I/O device.		
Pre-conditions	The Vibra I/O device must support setting frequency, per `SLVibraDescriptor::supportsFrequency`.		
Parameters	`self`	[in]	Interface self-reference.
	`pFrequency`	[out]	Address to store the vibration frequency. Range is [`SLVibraDescriptor::minFrequency`, `SLVibraDescriptor::maxFrequency`]

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		
See also	`SLVibraDescriptor` [see section 9.1.27].		

SetIntensity

```
SLresult (*SetIntensity) (
   SLVibraItf self,
   SLpermille intensity
);
```

Description	Sets the vibration intensity of the Vibra I/O device.		
Pre-conditions	The Vibra I/O device must support setting intensity, per `SLVibraDescriptor::supportsIntensity`.		
Parameters	`self`	[in]	Interface self-reference.
	`intensity`	[in]	Intensity of vibration. Range is [0, 1000].

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`
	`SL_RESULT_CONTROL_LOST`		

Comments	None.		
See also	`SLVibraDescriptor` [see section 9.1.27].		

GetIntensity			
`SLresult (*GetIntensity) ( const SLVibraItf self, SLpermille *pIntensity );`			
Description	Returns the vibration intensity of the Vibra I/O device.		
Pre-conditions	The Vibra I/O device must support setting intensity, per `SLVibraDescriptor::supportsIntensity`.		
Parameters	`self`	[in]	Interface self-reference.
	`pIntensity`	[out]	Address to store the vibration intensity of the Vibra I/O device. Range is [0, 1000].

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PRECONDITIONS_VIOLATED`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		
See also	`SLVibraDescriptor` [see section 9.1.27].		

8.46 SLVirtualizerItf

Description

This interface is for controlling audio virtualizer functionality. Audio virtualizer is a general name for an effect to spatialize audio channels. The exact behavior of this effect is dependent on the number of audio input channels and the types and number of audio output channels of the device. For example, in the case of a stereo input and stereo headphone output, a stereo widening effect is used when this effect is turned on. An input with 5.1-channels uses similarly a virtual surround algorithm. The exact behavior in each case is listed in the table below:

Table 13: Audio Virtualizer Functionality

	Mono Input	**Stereo Input**	**Multi-channel Input (more than two channels)**
Mono loudspeaker	No effect	No effect	No effect
Stereo loudspeakers	Pseudo-stereo	Stereo widening	Channel virtualization
Mono headset	No effect	No effect	No effect
Stereo headset	Pseudo-stereo	Stereo widening (Also other terms, including “Stereo virtualization,” are used for this kind of algorithm.)	Channel virtualization

As the table shows, the effect is not audible if mono output is used.

This interface affects different parts of the audio processing chain depending on which object the interface is exposed. If this interface is exposed on an Output Mix object, the effect is applied to the output mix. If this interface is exposed on a Player object, it is applied to the Player’s output only. For more information, see section 4.5.1.

When this interface is exposed on an Output Mix object, the effect will not affect the output of any of the 3D sources.

This interface is supported on the Output Mix [see section 7.9] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_VIRTUALIZER;

struct SLVirtualizerItf_;
typedef const struct SLVirtualizerItf_ * const * SLVirtualizerItf;

struct SLVirtualizerItf_ {
   SLresult (*SetEnabled)(
         SLVirtualizerItf self,
         SLboolean enabled
   );
   SLresult (*IsEnabled)(
         SLVirtualizerItf self,
         SLboolean *pEnabled
   );
   SLresult (*SetStrength)(
         SLVirtualizerItf self,
         SLpermille strength
   );
   SLresult (*GetRoundedStrength)(
         SLVirtualizerItf self,
         SLpermille *pStrength
   );
   SLresult (*IsStrengthSupported)(
         SLVirtualizerItf self,
         SLboolean *pSupported
   );
};
```

Interface ID

37cc2c00-dddd-11db-8577-0002a5d5c51b

Methods

SetEnabled

```
SLresult (*SetEnabled)(
   SLVirtualizerItf self,
   SLboolean enabled
);
```

Description	Enables the effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	enabled	[in]	True to turn on the effect, false to switch it off.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_CONTROL_LOST		
Comments	None		

IsEnabled

```
SLresult (*IsEnabled)(
   SLVirtualizerItf self,
   SLboolean *pEnabled
);
```

Description	Gets the enabled status of the effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pEnabled	[out]	True if the effect is on, false otherwise.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

SetStrength

```
SLresult (*SetStrength)(
    SLVirtualizerItf self,
    SLpermille strength
);
```

Description	Sets the strength of the effect. If the implementation does not support per mille accuracy for setting the strength, it is allowed to round the given strength to the nearest supported value. You can use the `GetRoundedStrength()` method to query the (possibly rounded) value that was actually set.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`strength`	[in]	Strength of the effect. The valid range for strength is [0, 1000], where 0 per mille designates the mildest effect and 1000 per mille designates the strongest effect,
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_CONTROL_LOST		
Comments	Please note that the strength does not affect the output if the effect is not enabled. This set method will in all cases store the setting, even if the effect is not enabled currently. Please note also that the strength can change if the output device is changed (as, for example, from stereo loudspeakers to stereo headphones) and those output devices use different algorithms with different accuracies. You can use device changed callbacks (`slAvailableAudioOutputsChangedCallback()` in the `SLAudioIODeviceCapabilitiesItf` interface) to monitor device changes and query the possibly-changed strength using `GetRoundedStrength()` in order, for example, for a graphical user interface to follow the current strength accurately.		
See Also	`SLAudioIODeviceCapabilitiesItf` [see section 8.12].		

GetRoundedStrength			
`SLresult (*GetRoundedStrength)( SLVirtualizerItf self, SLpermille *pStrength );`			
Description	Gets the current strength of the effect.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pStrength`	[out]	Strength of the effect. The valid range for strength is [0, 1000], where 0 per mille designates the mildest effect and 1000 per mille designates the strongest effect.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	Please note that the strength does not affect the output if the effect is not enabled. Please note also that in some cases the exact mapping of the strength to the underlying algorithms might not maintain the full accuracy exposed by the API. This is due to the fact that, for example, a global `VirtualizerItf` might actually internally control multiple algorithms that might use different accuracies: one for mono inputs and another for stereo inputs.		

IsStrengthSupported

```
SLresult (*IsStrengthSupported)(
    SLVirtualizerItf self,
    SLboolean *pSupported
);
```

Description	Tells whether setting strength is supported. If this method returns false, only one strength is supported and `SetStrength()` method always rounds to that value.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pSupported`	[out]	True if setting of the strength is supported, false if only one strength is supported.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.47 SLVisualizationItf

Description

This interface is for getting data for visualization purposes. The mechanism provides two kinds of data to the application:

- **Waveform data:** 512 consecutive 8-bit mono samples
- **Frequency data:** 256-length 8-bit magnitude FFT

The media object-specific visualization data is not affected by any of the API volume controls. Global visualization data is affected by all media object-specific OpenSL ES-controlled processing, such as volume and effects, but it is implementation-dependent whether global processing affects global visualization data.

This interface is optional on the Output Mix [see section 7.9] object, Audio Player [see section 7.2] object, Audio Recorder [see section 7.3] object, and MIDI Player [see section 7.8] object.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_VISUALIZATION;

struct SLVisualizationItf_;
typedef const struct SLVisualizationItf_ * const * SLVisualizationItf;

struct SLVisualizationItf_{
   SLresult (*RegisterVisualizationCallback)(
         SLVisualizationItf self,
         slVisualizationCallback callback,
         void *pContext,
         SLmilliHertz rate
   );
   SLresult (*GetMaxRate)(
         SLVisualizationItf self,
         SLmilliHertz* pRate
   );
};
```

Interface ID

e46b26a0-dddd-11db-8afd-0002a5d5c51b

Callbacks

slVisualizationCallback

```
typedef void (SLAPIENTRY *slVisualizationCallback) (
   SLVisualizationItf caller,
   void *pContext,
   const SLuint8 waveform[],
   const SLuint8 fft[],
   SLmilliHertz samplerate
);
```

Description	Gives visualization data to the application.		
Parameters	`caller`	[in]	Interface instantiation on which the callback was registered.
	`pContext`	[in]	User context data that is supplied when the callback method is registered.
	`waveform`	[in]	Waveform data, 512 consecutive 8-bit mono samples. The value range is [0, 255] and the value 128 represents a zero signal. The application must utilize the data before the callback returns; there is no guarantee that the pointer is valid after this function returns.
	`fft`	[in]	Frequency data, 256-length 8-bit magnitude FFT for visualization. The value range is [0, 255]. The same full-scale sine signal in both channels of a stereo signal produces a full-scale output at one of the FFT-bins, provided that the signal frequency matches bin-frequency exactly. The application must utilize the data before the callback returns; there is no guarantee that the pointer is valid after this function returns.
	`samplerate`	[in]	Sampling rate of the waveform and FFT data.
Comments	None.		
See also	`SLVisualization::RegisterVisualizationCallback`		

Methods

RegisterVisualizationCallback

```
SLresult (*RegisterVisualizationCallback)(
   SLVisualizationItf self,
   slVisualizationCallback callback,
   void *pContext,
   SLmilliHertz rate
);
```

Description	Sets or clears the `slVisualizationCallback`.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`callback`	[in]	Address of the callback.
	`pContext`	[in]	User context data that is supplied when the callback method is registered.
	`rate`	[in]	Rate how often the callback is called (in milliHertz). This parameter must always be greater than zero. Use `GetMaxRate()` method to query the maximum supported rate.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID SL_RESULT_MEMORY_FAILURE		
Comments	None.		

GetMaxRate

```
SLresult (*GetMaxRate)(
   SLVisualizationItf self,
   SLmilliHertz* pRate
);
```

Description	Gets the maximum supported rate. A valid implementation must return at least 20000 mHz.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pRate`	[out]	Maximum supported rate (in milliHertz) for how often the callback can be called.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		

8.48 SLVolumeItf

Description

This interface exposes controls for manipulating the object's audio volume properties.

This interface additionally exposes a stereo position control. Its exact effect is determined by the object's format; if the object's format is mono, a pan effect is applied, and if the object's format is stereo, a balance effect is applied.

This interface is supported on the Audio Player [see section 7.2], MIDI Player [see section 7.8] and Output Mix [see section 7.9] objects.

See Appendix C: for examples using this interface.

Prototype

```
SL_API extern const SLInterfaceID SL_IID_VOLUME;

struct SLVolumeItf_;
typedef const struct SLVolumeItf_ * const * SLVolumeItf;

struct SLVolumeItf_ {
   SLresult (*SetVolumeLevel) (
         SLVolumeItf self,
         SLmillibel level
   );
   SLresult (*GetVolumeLevel) (
         SLVolumeItf self,
         SLmillibel *pLevel
   );
   SLresult (*GetMaxVolumeLevel) (
         SLVolumeItf self,
         SLmillibel *pMaxLevel
   );
   SLresult (*SetMute) (
         SLVolumeItf self,
         SLboolean mute
   );
   SLresult (*GetMute) (
         SLVolumeItf self,
         SLboolean *pMute
   );
   SLresult (*EnableStereoPosition) (
         SLVolumeItf self,
         SLboolean enable
   );
   SLresult (*IsEnabledStereoPosition) (
         SLVolumeItf self,
         SLboolean *pEnable
   );
```

```
      SLresult (*SetStereoPosition) (
            SLVolumeItf self,
            SLpermille stereoPosition
      );
      SLresult (*GetStereoPosition) (
            SLVolumeItf self,
            SLpermille *pStereoPosition
      );
   };
```

Interface ID

09e8ede0-ddde-11db-b4f6-0002a5d5c51b

Defaults

- Volume level: 0 mB
- Mute: disabled (not muted)
- Stereo position: disabled, 0 ‰ (center)

Methods

SetVolumeLevel

```
SLresult (*SetVolumeLevel) (
   SLVolumeItf self,
   SLmillibel level
);
```

Description	Sets the object's volume level.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	level	[in]	Volume level in millibels. The valid range is [SL_MILLIBEL_MIN, maximum supported level], where maximum supported level can be queried with the method GetMaxVolumeLevel(). The maximum supported level is always at least 0 mB.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	If the object is muted, calls to SetVolumeLevel() will still change the internal volume level, but this will have no audible effect until the object is unmuted.		
See also	SetMute()		

GetVolumeLevel

```
SLresult (*GetVolumeLevel) (
   SLVolumeItf self,
   SLmillibel *pLevel
);
```

Description	Gets the object's volume level.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pLevel	[out]	Pointer to a location to receive the object's volume level in millibels. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

GetMaxVolumeLevel

```
SLresult (*GetMaxVolumeLevel) (
   SLVolumeItf self,
   SLmillibel *pMaxLevel
);
```

Description	Gets the maximum supported level.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMaxLevel	[out]	Pointer to a location to receive the maximum supported volume level in millibels. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The maximum supported level is implementation-dependent, but will always be at least 0 mB.		
See also	None.		

SetMute

```
SLresult (*SetMute) (
   SLVolumeItf self,
   SLboolean mute
);
```

Description	Mutes or unmutes the object.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	mute	[in]	If true, the object is muted. If false, the object is unmuted.
Return value	The return value can be the following: SL_RESULT_SUCCESS		
Comments	Muting the object does not change the volume level reported by GetVolumeLevel(). Calling SetMute() with mute set to true when the object is already muted is a valid operation that has no effect. Calling SetMute() with mute set to false when the object is already unmuted is a valid operation that has no effect.		
See also	GetVolumeLevel()		

GetMute

```
SLresult (*GetMute) (
   SLVolumeItf self,
   SLboolean *pMute
);
```

Description	Retrieves the object's mute state.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	pMute	[out]	Pointer to a boolean to receive the object's mute state. This must be non-NULL.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	None.		
See also	None.		

EnableStereoPosition

```
SLresult (*EnableStereoPosition) (
SLVolumeItf self,
SLboolean enable
);
```

Description	Enables or disables the stereo positioning effect.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`enable`	[in]	If true, enables the stereo position effect. If false, disables the stereo positioning effect (no attenuation due to stereo positioning is applied to the left or right channels).
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	If the output of the object is monophonic this setting doesn't have any effect. This method does not affect the stereo position nor its availability through `GetStereoPosition()` and `SetStereoPosition()`. **PROFILE NOTES** *This method is mandated only in the Game and Music profiles.*		
See also	None.		

IsEnabledStereoPosition

```
SLresult (*IsEnabledStereoPosition) (
   const SLVolumeItf self,
   SLboolean *pEnable
);
```

Description	Returns the enabled state of the stereo positioning effect.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pEnable`	[out]	Pointer to a location to receive the enabled state of the stereo positioning effect.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	If the output of the object is monophonic this setting doesn't have any effect. **PROFILE NOTES** *This method is mandated only in the Game and Music profiles.*		
See also	None.		

SetStereoPosition

```
SLresult (*SetStereoPosition) (
    SLVolumeItf self,
    SLpermille stereoPosition
);
```

Description	Sets the stereo position of the object; For mono objects, this will control a constant energy pan effect, and for stereo objects, this will control a balance effect.		
Pre-conditions	None.		
Parameters	self	[in]	Interface self-reference.
	stereoPosition	[in]	Stereo position in the range [-1000 ‰, 1000 ‰]. A stereo position of 0 ‰ indicates the object is in the center. That is, in the case of balance, no attenuation is applied to the left and right channels; and in the case of pan, 3 dB attenuation is applied to the left and right channels. A stereo position of –1000 ‰ pans the object fully to the left; the right channel is silent. A stereo position of 1000 ‰ pans the object fully to the right; the left channel is silent.
Return value	The return value can be one of the following: SL_RESULT_SUCCESS SL_RESULT_PARAMETER_INVALID		
Comments	The exact pan and balance curves used for this method are implementation-dependent, subject to satisfying the parameter description. For objects whose (input) format is mono, this method controls a constant energy pan effect. For objects whose (input) format is stereo, this method controls a balance effect. If the output of the object is monophonic this setting doesn't have any effect. **PROFILE NOTES** *This method is mandated only in the Game and Music profiles.*		
See also	None.		

GetStereoPosition

```
SLresult (*GetStereoPosition) (
   const SLVolumeItf self,
   SLpermille *pStereoPosition
);
```

Description	Gets the object's stereo position setting.		
Pre-conditions	None.		
Parameters	`self`	[in]	Interface self-reference.
	`pStereoPosition`	[out]	Pointer to a location to receive the current stereo position setting. This must be non-`NULL`.

Return value	The return value can be one of the following:
	`SL_RESULT_SUCCESS`
	`SL_RESULT_PARAMETER_INVALID`		

Comments	None.		
See also	None.		

9 Macros and Typedefs

9.1 Structures

9.1.1 SLAudioCodecDescriptor

```
typedef struct SLAudioCodecDescriptor_ {
  SLuint32     maxChannels;
  SLuint32     minBitsPerSample;
  SLuint32     maxBitsPerSample;
  SLmilliHertz minSampleRate;
  SLmilliHertz maxSampleRate;
  SLboolean    isFreqRangeContinuous;
  SLmilliHertz *pSampleRatesSupported;
  SLuint32     numSampleRatesSupported;
  SLuint32     minBitRate;
  SLuint32     maxBitRate;
  SLboolean    isBitrateRangeContinuous;
  SLuint32     *pBitratesSupported;
  SLuint32     numBitratesSupported;
  SLuint32     profileSetting;
  SLuint32     modeSetting;
  SLuint32     streamFormat;
} SLAudioCodecDescriptor;
```

This structure is used for querying the capabilities of an audio codec.

Field	Description
maxChannels	Maximum number of audio channels.
minBitsPerSample	Minimum bits per sample of PCM data.
maxBitsPerSample	Maximum bits per sample of PCM data.
minSampleRate	Minimum sampling rate supported.
maxSampleRate	Maximum sampling rate supported.
isFreqRangeContinuous	Returns SL_BOOLEAN_TRUE if the device supports a continuous range of sampling rates between minSampleRate and maxSampleRate; otherwise returns SL_BOOLEAN_FALSE.
pSampleRatesSupported	Indexed array containing the supported sampling rates. Ignored if isFreqRangeContinuous is SL_BOOLEAN_TRUE. If pSampleRatesSupported is NULL, the number of supported sample rates is returned in numSampleRatesSupported.
numSampleRatesSupported	Size of the pSamplingRatesSupported array. Ignored if isFreqRangeContinuous is SL_BOOLEAN_TRUE.

Field	Description
minBitRate	Minimum bit-rate.
maxBitRate	Maximum bit-rate.
isBitrateRangeContinuous	Returns SL_BOOLEAN_TRUE if the device supports a continuous range of bitrates between minBitRate and maxBitRate; otherwise returns SL_BOOLEAN_FALSE.
pBitratesSupported	Indexed array containing the supported bitrates. Ignored if isBitrateRangeContinuous is SL_BOOLEAN_TRUE. If pBitratesSupported is NULL, the number of supported bitrates is returned in numBitratesSupported.
numBitratesSupported	Size of the pBitratesSupported array. Ignored if isBitrateRangeContinuous is SL_BOOLEAN_TRUE.
profileSetting	Profile supported by codec. See SL_AUDIOPROFILE defines [see section 9.2.3].
modeSetting	Level supported by codec. See SL_AUDIOMODE defines [see section 9.2.3].
streamFormat	Stream format of the codec. See SL_AUDIOSTREAMFORMAT defines [see section 9.2.3].

9.1.2 SLAudioEncoderSettings

```
typedef struct SLAudioEncoderSettings_ {
  SLuint32 encoderId;
  SLuint32 channelsIn;
  SLuint32 channelsOut;
  SLmilliHertz sampleRate;
  SLuint32 bitRate;
  SLuint32 bitsPerSample;
  SLuint32 rateControl;
  SLuint32 profileSetting;
  SLuint32 levelSetting;
  SLuint32 channelMode;
  SLuint32 streamFormat;
  SLuint32 encodeOptions;
  SLuint32 blockAlignment;
} SLAudioEncoderSettings;
```

This structure is used to set the audio encoding parameters. Set a field's value to zero to specify use of the default setting for that encoding parameter.

Field	Description
encoderId	Identifies the supported audio encoder. Refer to SL_AUDIOOCODEC defines in section 9.2.1.

Field	**Description**
`channelsIn`	Number of input audio channels.
`channelsOut`	Number of output channels in encoded data. In case of contradiction between this field and the `channelMode` field, the `channelMode` field overrides.
`sampleRate`	Audio sample rate of input audio data in millihertz
`bitRate`	Bitrate of encoded data.
`bitsPerSample`	Bits per sample of input data.
`rateControl`	Encoding rate control mode. See SL_RATECONTROLMODE defines in section 9.2.41.
`profileSetting`	Profile to use for encoding. See SL_AUDIOPROFILE defines in section 9.2.3.
`levelSetting`	Level to use for encoding. See SL_AUDIOMODE defines in section 9.2.3.
`channelMode`	Channel mode for encoder. See SL_AUDIOCHANMODE defines in section 9.2.3.
`streamFormat`	Format of encoded bit-stream. For example, AMR encoders use this to select between IF1, IF2, or RTPPAYLOAD bit-stream formats. Refer to SL_AUDIOSTREAMFORMAT_XXX defines in section 9.2.3.
`encodeOptions`	Codec specific encoder options. For example, WMA encoders use it to specify codec version, framesize, frequency extensions, and other options. See the relevant encoder documentation for format. This is typically a bitfield specifying encode options. Use a value of zero to specify use of the default encoder settings for the encoder.
`blockAlignment`	Block alignment in bytes of an audio sample.

9.1.3 SLAudioInputDescriptor

```
typedef struct SLAudioInputDescriptor_ {
   SLchar * pDeviceName;
   SLint16 deviceNameLength;
   SLint16 deviceConnection;
   SLint16 deviceScope;
   SLint16 deviceLocation;
   SLboolean isForTelephony;
   SLmilliHertz minSampleRate;
   SLmilliHertz maxSampleRate;
   SLboolean isFreqRangeContinuous;
   SLmilliHertz * pSamplingRatesSupported;
   SLint16 numOfSamplingRatesSupported;
   SLint16 maxChannels;
} SLAudioInputDescriptor;
```

This structure is used for returning the description of audio input device capabilities. The deviceConnection, deviceScope and deviceLocation fields collectively describe the type of audio input device in a standardized way, while still allowing new device types to be added (by vendor-specific extensions of the corresponding macros), if necessary. For example, on a mobile phone, the integrated microphone would have the following values for each of these three fields, respectively: SL_DEVCONNECTION_INTEGRATED, SL_DEVSCOPE_USER and SL_DEVLOCATION_HANDSET, while a Bluetooth headset microphone would have the following values: SL_DEVCONNECTION_ATTACHED_WIRELESS, SL_DEVSCOPE_USER and SL_DEVLOCATION_HEADSET.

Field	Description
pDeviceName	Human-readable string representing the name of the device, such as "Bluetooth microphone" or "wired microphone."
deviceNameLength	Length of the array for pDeviceName.
deviceConnection	One of the device connection types listed in the SL_DEVCONNECTION macros.
deviceScope	One of the device scope types listed in the SL_DEVSCOPE macros.
deviceLocation	One of the device location types listed in the SL_DEVLOCATION macros
isForTelephony	Returns SL_BOOLEAN_TRUE if the audio input device is deemed suitable for telephony uplink audio; otherwise returns SL_BOOLEAN_FALSE. For example: a line-in jack would not be considered suitable for telephony, as it is difficult to determine what can be connected to it.
minSampleRate	Minimum sampling rate supported.
maxSampleRate	Maximum sampling rate supported.
isFreqRangeContinuous	Returns SL_BOOLEAN_TRUE if the input device supports a continuous range of sampling rates between minSampleRate and maxSampleRate; otherwise returns SL_BOOLEAN_FALSE.
pSamplingRatesSupported	Indexed array containing the supported sampling rates, as defined in the SL_SAMPLING_RATE macros. Ignored if isFreqRangeContinuous is SL_BOOLEAN_TRUE.
numOfSamplingRatesSupported	Size of the samplingRatesSupported array. Ignored if isFreqRangeContinuous is SL_BOOLEAN_TRUE.
maxChannels	Maximum number of channels supported; for mono devices, value would be 1.

The table below shows examples of the first five fields of the SLAudioInputDescriptor struct for various audio input devices. For the sake of brevity and clarity, the full names of the SL_DEV macros have been abbreviated to include just the distinct portion of the names (such as SL_DEVCONNECTION_INTEGRATED appears as INTEGRATED and SL_DEVSCOPE_PRIVATE as PRIVATE).

Table 14: Examples of Audio Input Devices

deviceName	device-Connection	device-Scope	device-Location	isForTelephony
Handset microphone	INTEGRATED	USER	HANDSET	TRUE
Bluetooth microphone	WIRELESS	USER	HEADSET	TRUE
Wired headset microphone	WIRED	USER	HEADSET	TRUE
Carkit microphone	WIRED	ENVIRO-NMENT	CARKIT	TRUE
Carkit handset microphone	WIRED	USER	CARKIT	TRUE
System line-in jack	INTEGRATED	UNKNOWN	HANDSET	FALSE
Networked Media Server	NETWORK	UNKNOWN	RESLTE	FALSE

9.1.4 SLAudioOutputDescriptor

```
typedef struct SLAudioOutputDescriptor_ {
   SLchar *pDeviceName;
   SLint16 deviceNameLength;
   SLint16 deviceConnection;
   SLint16 deviceScope;
   SLint16 deviceLocation;
   SLboolean isForTelephony;
   SLmilliHertz minSampleRate;
   SLmilliHertz maxSampleRate;
   SLboolean isFreqRangeContinuous;
   SLmilliHertz *pSamplingRatesSupported;
   SLint16 numOfSamplingRatesSupported;
   SLint16 maxChannels;
} SLAudioOutputDescriptor;
```

This structure is used for returning the description of audio output device capabilities. The deviceConnection, deviceScope and deviceLocation fields collectively describe the type of audio output device in a standardized way, while still allowing new device types to be added (by vendor-specific extensions of the corresponding macros), if necessary. For example, on a mobile phone, the earpiece would have the following values for each of these three fields, respectively: SL_DEVCONNECTION_INTEGRATED, SL_DEVSCOPE_USER and

SL_DEVLOCATION_HANDSET, while a pair of speakers that are part of a music dock would have the following: SL_DEVCONNECTION_ATTACHED_WIRED, SL_DEVSCOPE_ENVIRONMENT and SL_DEVLOCATION_DOCK.

Field	**Description**
pDeviceName	Human-readable string representing the name of the output device, such as “integrated loudspeaker” or “Bluetooth headset.”
deviceNameLength	Length of the array for pDeviceName.
deviceConnection	One of the device connection types listed in the SL_DEVCONNECTION macros.
deviceScope	One of the device scope types listed in the SL_DEVSCOPE macros.
deviceLocation	One of the device location types listed in the SL_DEVLOCATION macros.
isForTelephony	Returns SL_BOOLEAN_TRUE if the audio output device is deemed suitable for telephony downlink audio; otherwise returns SL_BOOLEAN_FALSE. For example, a line-out jack would not be a suitable for telephony downlink audio.
minSampleRate	Minimum sampling rate supported.
maxSampleRate	Maximum sampling rate supported.
isFreqRangeContinuous	Returns SL_BOOLEAN_TRUE if the output device supports a continuous range of sampling rates between minSampleRate and maxSampleRate; otherwise returns SL_BOOLEAN_FALSE.
pSamplingRatesSupported	Indexed array containing the supported sampling rates, as defined in the SL_SAMPLINGRATE macros. Ignored if isFreqRangeContinuous is SL_BOOLEAN_TRUE.
numOfSamplingRatesSupported	Size of the samplingRatesSupported array. Ignored if isFreqRangeContinuous is SL_BOOLEAN_TRUE.
maxChannels	Maximum number of channels supported; for mono devices, value would be 1.

The table below shows examples of the first six fields of the SLAudioOutputDescriptor struct for various audio output devices. For the sake of brevity and clarity, the full names of the SL_DEV macros have been abbreviated to include just the distinct portion of the names (e.g. SL_DEVCONNECTION_INTEGRATED appears as INTEGRATED and SL_DEVSCOPE_USER as USER).

Table 15: Examples of Audio Output Devices

deviceName	device-Connection	device-Scope	device-Location	isFor-Telephony
Earpiece	INTEGRATED	USER	HANDSET	TRUE
Loudspeaker	INTEGRATED	ENVIRO-MENT	HANDSET	TRUE
Bluetooth headset speaker	WIRELESS	USER	HEADSET	TRUE
Wired headset speaker	WIRED	USER	HEADSET	TRUE
Carkit loudspeaker	WIRED	ENVIRO-MENT	CARKIT	TRUE
Carkit handset speaker	WIRED	USER	CARKIT	TRUE
System line-out jack	INTEGRATED	UNKNOWN	HANDSET	FALSE
Dock loudspeaker	WIRED	ENVIRON-MENT	DOCK	FALSE
FM Radio Transmitter	WIRED	ENVIRON-MENT	DOCK	FALSE
Networked media renderer	NETWORK	UNKNOWN	REMOTE	FALSE

9.1.5 SLBufferQueueState

```
typedef struct SLBufferQueueState_ {
   SLuint32    count;
   SLuint32    index;
} SLBufferQueueState;
```

Field	Description
`count`	Number of buffers currently in the queue
`index`	Index of the currently playing or filling buffer. This is a linear index that keeps a cumulative count of the number of buffers played. It is set to zero by a call to the `SLBufferQueue::Clear()` method [see section 8.14].

9.1.6 SLDataFormat_MIME

```
typedef struct SLDataFormat_MIME_ {
   SLuint32          formatType;
   const SLchar *    pMimeType;
   SLuint32          containerType;
} SLDataFormat_MIME;
```

Fields include:

Field	Description
formatType	The format type, which must always be SL_DATAFORMAT_MIME for this structure.
pMimeType	The mime type of the data expressed as a string.
containerType	The container type of the data. When an application uses this structure to specify the data source for a player use case, the application may leave the containerType unspecified (for example SL_CONTAINERTYPE_UNSPECIFIED) or may provide a specific value as a hint to the player. When an application uses this structure to specify the data sink for a recorder use case, the application is dictating the container type of the captured content.

9.1.7 SLDataFormat_PCM

NOTE: This structure is deprecated. Use SLDataFormat_PCM_EX instead.

```
typedef struct SLDataFormat_PCM_ {
   SLuint32          formatType;
   SLuint32          numChannels;
   SLuint32          samplesPerSec;
   SLuint32          bitsPerSample;
   SLuint32          containerSize;
   SLuint32          channelMask;
   SLuint32          endianness;
} SLDataFormat_PCM;
```

Fields include:

Field	Description
formatType	The format type, which must always be SL_DATAFORMAT_PCM for this structure.
numChannels	Numbers of audio channels present in the data. Multi-channel audio is always interleaved in the data buffer.
samplesPerSec	The audio sample rate of the data in milliHertz. **Note: This is set to milliHertz and not Hertz, as the field name would suggest.**

Field	Description
bitsPerSample	Number of actual data bits in a sample. If bitsPerSample is equal to 8 then the data's representation is SL_PCM_REPRESENTATION_UNSIGNED_INT. Otherwise, the data's representation is SL_PCM_REPRESENTATION_SIGNED_INT.
containerSize	The container size for PCM data in bits, for example 24 bit data in a 32 bit container. Data is left-justified within the container. For best performance, it is recommended that the container size be the size of the native data types.
channelMask	Channel mask indicating mapping of audio channels to speaker location. The channelMask member specifies which channels are present in the multichannel stream. The least significant bit corresponds to the front left speaker (SL_SPEAKER_FRONT_LEFT), the next least significant bit corresponds to the front right speaker (SL_SPEAKER_FRONT_RIGHT), and so on. The full list of valid speaker locations is defined in section 9.2.50. The channels specified in channelMask must be present in the prescribed order (from least significant bit up). For example, if only SL_SPEAKER_FRONT_LEFT and SL_SPEAKER_FRONT_RIGHT are specified, the samples for the front left speaker must come first in the interleaved stream. The number of bits set in channelMask should be the same as the number of channels specified in numChannels with the caveat that a default setting of zero indicates stereo format (i.e. the setting is equivalent to SL_SPEAKER_FRONT_LEFT | SL_SPEAKER_FRONT_RIGHT)
endianness	Endianness of the audio data. See SL_BYTEORDER macro for definition [see section 9.2.7].

9.1.8 SLDataFormat_PCM_EX

```
typedef struct SLDataFormat_PCM_EX_ {
   SLuint32          formatType;
   SLuint32          numChannels;
   SLuint32          sampleRate;
   SLuint32          bitsPerSample;
   SLuint32          containerSize;
   SLuint32          channelMask;
   SLuint32          endianness;
   SLuint32          representation;
} SLDataFormat_PCM_EX;
```

Fields include:

Field	Description
formatType	The format type, which must always be SL_DATAFORMAT_PCM_EX for this structure.

Field	Description
numChannels	Numbers of audio channels present in the data. Multi-channel audio is always interleaved in the data buffer.
sampleRate	The audio sample rate of the data in milliHertz.
bitsPerSample	Number of actual data bits in a sample.
containerSize	The container size for PCM data in bits, for example 24 bit data in a 32 bit container. Data is left-justified within the container. For best performance, it is recommended that the container size be the size of the native data types.
channelMask	Channel mask indicating mapping of audio channels to speaker location. The channelMask member specifies which channels are present in the multichannel stream. The least significant bit corresponds to the front left speaker (SL_SPEAKER_FRONT_LEFT), the next least significant bit corresponds to the front right speaker (SL_SPEAKER_FRONT_RIGHT), and so on. The full list of valid speaker locations is defined in section 9.2.47. The channels specified in channelMask must be present in the prescribed order (from least significant bit up). For example, if only SL_SPEAKER_FRONT_LEFT and SL_SPEAKER_FRONT_RIGHT are specified, the samples for the front left speaker must come first in the interleaved stream. The number of bits set in channelMask should be the same as the number of channels specified in numChannels with the caveat that a default setting of zero indicates stereo format (i.e. the setting is equivalent to SL_SPEAKER_FRONT_LEFT | SL_SPEAKER_FRONT_RIGHT)
endianness	Endianness of the audio data. See SL_BYTEORDER macro for definition [see section 9.2.7]
representation	Representation of the audio data. See SL_PCM_REPRESENTATION macro for definition [see section 9.2.33].

9.1.9 SLDataLocator_Address

```
typedef struct SLDataLocator_Address_ {
   SLuint32    locatorType;
   void        *pAddress;
   SLuint32    length;
} SLDataLocator_Address;
```

Fields include:

Field	Description
locatorType	Locator type, which must always be SL_DATALOCATOR_ADDRESS for this structure.

Field	**Description**
pAddress	Address of the first byte of data.
Length	Length of the data in bytes.

9.1.10 SLDataLocator_IODevice

```
typedef struct SLDataLocator_IODevice_ {
   SLuint32    locatorType;
   SLuint32    deviceType;
   SLuint32    deviceID;
   SLObjectItf device;
} SLDataLocator_IODevice;
```

Fields include:

Field	**Description**
locatorType	Locator type, which must be SL_DATALOCATOR_IODEVICE for this structure [see section 9.2.12].
deviceType	Type of I/O device. See SL_IODEVICE macros in section 9.2.21.
deviceID	ID of the device. Ignored if device is not NULL.
device	I/O device object itself. Must be NULL if deviceID parameter is to be used.

9.1.11 SLDataLocator_BufferQueue

```
typedef struct SLDataLocator_BufferQueue_ {
   SLuint32    locatorType;
   SLuint32    numBuffers;
} SLDataLocator_BufferQueue;
```

Struct representing a data locator for a buffer queue.

Field	Description
locatorType	Locator type, which must be SL_DATALOCATOR_BUFFERQUEUE for this structure [see section 9.2.12].
numBuffers	Number of buffers to allocate for buffer queue.

9.1.12 SLDataLocator_ContentPipe

```
typedef struct SLDataLocator_ContentPipe_ {
    SLuint32 locatorType;
    void * pContentPipe;
    const SLchar * pURI;
} SLDataLocator_ContentPipe;
```

Struct representing a data locator for a content pipe.

Field	Description
locatorType	Locator type, which must be SL_DATALOCATOR_CONTENTPIPE for this structure [see section 9.2.12].
pContentPipe	Pointer to the structure for the content pipe. Refer to the Content Pipe Specification [CP] for details.
pURI	The URI for the content pipe.

9.1.13 SLDataLocator_MediaObject

```
typedef struct SLDataLocator_MediaObject_ {
    SLuint32 locatorType;
    SLObjectItf mediaObject;
} SLDataLocator_MediaObject;
```

Struct representing a data locator for a media object.

Field	Description
locatorType	Locator type, which must be SL_DATALOCATOR_MEDIAOBJECT for this structure [see section 9.2.12].
mediaObject	Media object created by the engine.

9.1.14 SLDataLocator_MIDIBufferQueue

```
typedef struct SLDataLocator_MIDIBufferQueue_ {
   SLuint32    locatorType;
   SLuint32    tpqn;
   SLuint32    numBuffers;
} SLDataLocator_MIDIBufferQueue;
```

Struct representing a data locator for a MIDI buffer queue.

Field	Description
locatorType	Locator type, which must be SL_DATALOCATOR_MIDIBUFFERQUEUE for this structure [see section 9.2.12].
tpqn	MIDI ticks per quarter note (ticks per beat / pulses per quarter note). Range is [0, 32767].
numBuffers	Number of buffers to allocate for buffer queue

9.1.15 SLDataLocator_Null

```
typedef struct SLDataLocator_Null_ {
    SLuint32 locatorType;
} SLDataLocator_Null;
```

Struct representing a null data locator. This is used in conjuction with the `SLDynamicSourceSinkChangeItf` interface.

Field	Description
`locatorType`	Locator type, which must be `SL_DATALOCATOR_NULL` for this structure [see section 9.2.12].

9.1.16 SLDataLocator_OutputMix

```
typedef struct SLDataLocator_OutputMix_ {
   SLuint32          locatorType;
   SLObjectItf       outputMix;
} SLDataLocator_OutputMix;
```

Fields include:

Field	Description
`locatorType`	Locator type, which must be `SL_DATALOCATOR_OUTPUTMIX` for this structure [see section 9.2.12].
`outputMix`	The OutputMix object as retrieved from the engine.

9.1.17 SLDataLocator_URI

```
typedef struct SLDataLocator_URI_ {
   SLuint32          locatorType;
   const SLchar *    pURI;
} SLDataLocator_URI;
```

Fields include:

Field	Description
`locatorType`	Locator type, which must always be `SL_DATALOCATOR_URI` for this structure [see section 9.2.12].
`pURI`	URI expressed as a string.

9.1.18 SLDataSink

```
typedef struct SLDataSink_ {
   void *pLocator;
   void *pFormat;
} SLDataSink;
```

Fields include:

Field	Description
pLocator	Pointer to the specified data locator structure. This may point to any of the following structures: SLDataLocator_Address SLDataLocator_BufferQueue SLDataLocator_IODevice (LED & Vibra only) SLDataLocator_MediaObject SLDataLocator_Null SLDataLocator_OutputMix SLDataLocator_URI SLDataLocator_ContentPipe The first field of each of these structures includes the 32 bit locatorType field, which identifies the locator type (see SL_DATALOCATOR definitions in section 9.2.12) and hence the structure pointed to. The locator SLDataLocator_OutputMix is used in the common case where a player's output should be directed to the default audio output mix. Note: The available SL_DATALOCATOR definitions may be extended through an API extension.
pFormat	A pointer to the specified format structure. This may point to any of the following structures. SLDataFormat_PCM **(Deprecated)** SLDataFormat_PCM_EX SLDataFormat_MIME The first field of each of these structures includes the 32 bit formatType field, which identifies the format type (SL_DATAFORMAT definitions [see section 9.2.11]) and hence the structure pointed to. This parameter is ignored if pLocator is SLDataLocator_IODevice, SLDataLocator_OutputMix, SLDataLocator_MediaObject, or SLDataLocator_Null

9.1.19 SLDataSource

```
typedef struct SLDataSource_ {
   void *pLocator;
   void *pFormat;
} SLDataSource;
```

Fields include:

Field	Description
pLocator	Pointer to the specified data locator structure. This may point to any of the following structures. SLDataLocator_Address SLDataLocator_BufferQueue SLDataLocator_IODevice SLDataLocator_MediaObject SLDataLocator_MIDIBufferQueue SLDataLocator_Null SLDataLocator_URI SLDataLocator_ContentPipe The first field of each of these structures includes the 32 bit locatorType field, which identifies the locator type (see SL_DATALOCATOR definitions) and hence the structure pointed to. Note: The available SL_DATALOCATOR definitions may be extended through an API extension.
pFormat	A pointer to the specified format structure. This may point to any of the following structures. SLDataFormat_PCM **(Deprecated)** SLDataFormat_PCM_EX SLDataFormat_MIME The first field of each of these structure includes the 32 bit formatType field, which identifies the format type (SL_DATAFORMAT definitions) and hence the structure pointed to. This parameter is ignored if pLocator is SLDataLocator_IODevice, SLDataLocator_MediaObject, or SLDataLocator_Null.

9.1.20 SLEngineOption

```
typedef struct SLEngineOption_ {
   SLuint32 feature;
   SLuint32 data;
} SLEngineOption;
```

Structure used for specifying different options during engine creation.

Field	Description
feature	Feature identifier. See SL_ENGINEOPTION macros [see section 9.2.18].
data	Value to use for feature.

9.1.21 SLEnvironmentalReverbSettings

```
typedef struct SLEnvironmentalReverbSettings_ {
   SLmillibel roomLevel;
   SLmillibel roomHFLevel;
   SLmillisecond decayTime;
   SLpermille decayHFRatio;
   SLmillibel reflectionsLevel;
   SLmillisecond reflectionsDelay;
   SLmillibel reverbLevel;
   SLmillisecond reverbDelay;
   SLpermille diffusion;
   SLpermille density;
} SLEnvironmentalReverbSettings;
```

This structure can store all the environmental reverb settings. The meaning of the parameters is defined in the SLEnvironmentalReverbItf interface [see section 8.23].

Field	Description
roomLevel	Environment's volume level in millibels. The valid range is [SL_MILLIBEL_MIN, 0].
roomHFLevel	High-frequency attenuation level in millibels. The valid range is [SL_MILLIBEL_MIN, 0].
decayTime	Decay time in milliseconds. The valid range is [100, 20000].
decayHFRatio	Relative decay time at the high-frequency reference (5 kHz) using a permille scale. The valid range is [100, 2000].
reflectionsLevel	Early reflections attenuation level in millibels. The valid range is [SL_MILLIBEL_MIN, 1000].
reflectionsDelay	Early reflections delay length in milliseconds. The valid range is [0, 300].
reverbLevel	Late reverb level in millibels. The valid range is [SL_MILLIBEL_MIN, 2000].

Field	Description
reverbDelay	Late reverb delay in milliseconds. The valid range is [0, 100].
diffusion	Diffusion level expressed in permilles. The valid range is [0, 1000].
density	Density level expressed in permilles. The valid range is [0, 1000].

9.1.22 SLHSL

```
typedef struct SLHSL_ {
  SLmillidegree hue;
  SLpermille   saturation;
  SLpermille   lightness;
} SLHSL;
```

SLHSL represents a color defined in terms of the HSL color space.

Field	Description
hue	Hue. Range is [0, 360000] in millidegrees. (Refers to the range between 0 and 360 degrees).
saturation	Saturation of the color. Range is [0, 1000] in permille. (Refers to the range between 0.0% and 100.0%).
lightness	Lightness of the color. Range is [0, 1000] in permille. (Refers to the range between 0.0% and 100.0%).

9.1.23 SLInterfaceID

```
typedef const struct SLInterfaceID_ {
  SLuint32 time_low;
  SLuint16 time_mid;
  SLuint16 time_hi_and_version;
  SLuint16 clock_seq;
  SLuint8 node[6];
} * SLInterfaceID;
```

The interface ID type.

Field	Description
time_low	Low field of the timestamp.
time_mid	Middle field of the timestamp.
time_hi_and_version	High field of the timestamp multiplexed with the version number.
clock_seq	Clock sequence.
node	Spatially unique node identifier.

9.1.24 SLLEDDescriptor

```
typedef struct SLLEDDescriptor_ {
   SLuint8  ledCount;
   SLuint8  primaryLED;
   SLuint32 colorMask;
} SLLEDDescriptor;
```

SLLEDDescriptor represents the capabilities of the LED array I/O Device.

Field	Description
ledCount	Number of LEDs in the array. Range is [1, 32].
primaryLED	Index of the primary LED, which is the main status LED of the device. Range is [0, ledCount-1].
colorMask	Bitmask indicating which LEDs support color. Valid bits range from the least significant bit, which indicates the first LED in the array, to bit ledCount–1, which indicates the last LED in the array.

9.1.25 SLMetadataInfo

```
typedef struct SLMetadataInfo_ {
  SLuint32   size;
  SLuint32   encoding;
  SLchar langCountry[16];
  SLuint8   data[1];
} SLMetadataInfo;
```

SLMetadataInfo represents a key or a value from a metadata item key/value pair.

Field	Description
size	Size of the data in bytes. size must be greater than 0.
encoding	Character encoding of the data.
langCountry	Language / country code of the data (see note below).
data	Key or value, as represented by the encoding.

The language / country code may be a language code, a language / country code, or a country code. When specifying the code, note that a partially-specified code will match fully-specified codes that match the part that is specified. For example, “en” will match “en-us” and other “en” variants. Likewise, “us” will match “en-us” and other “us” variants.

Formatting of language codes and language / country codes is defined by IETF RFC 3066 [RFC3066] (which incorporates underlying ISO specifications 639 [ISO639] and 3166 [ISO3166] and a syntax). Formatting of country codes is defined by ISO 3166 [ISO3166].

9.1.26 SLVec3D

```
typedef struct SLVec3D_ {
   SLint32     x;
   SLint32     y;
   SLint32     z;
} SLVec3D;
```

This structure is used for representing coordinates in Cartesian form, see section 4.4.2.1.

Field	Description
x	x-axis coordinate.
y	y-axis coordinate.
z	z-axis coordinate.

9.1.27 SLVibraDescriptor

```
typedef struct SLVibraDescriptor_ {
   SLboolean supportsFrequency;
   SLboolean supportsIntensity;
   SLmilliHertz minFrequency;
   SLmilliHertz maxFrequency;
} SLVibraDescriptor;
```

`SLVibraDescriptor` represents the capabilities of the Vibra I/O device.

Field	Description
supportsFrequency	Boolean indicating whether the Vibra I/O device supports setting the frequency of vibration.
supportsIntensity	Boolean indicating whether the Vibra I/O device supports setting the intensity of vibration.
minFrequency	Minimum frequency supported by the Vibra I/O device. Range is [1, SL_MILLIHERTZ_MAX]. If supportsFrequency is set to SL_BOOLEAN_FALSE, this will be set to 0.
maxFrequency	Maximum frequency supported by the Vibra I/O device. Range is [minFrequency, SL_MILLIHERTZ_MAX]. If supportsFrequency is set to SL_BOOLEAN_FALSE, this will be set to 0.

9.2 Macros

9.2.1 SL_3DHINT

```
#define SL_3DHINT_OFF                      ((SLuint16) 0x0000)
#define SL_3DHINT_QUALITY_LOWEST           ((SLuint16) 0x0001)
#define SL_3DHINT_QUALITY_LOW              ((SLuint16) 0x4000)
#define SL_3DHINT_QUALITY_MEDIUM           ((SLuint16) 0x8000)
#define SL_3DHINT_QUALITY_HIGH             ((SLuint16) 0xC000)
#define SL_3DHINT_QUALITY_HIGHEST          ((SLuint16) 0xFFFF)
```

These macros are used to define a 3D source or group's importance.

Value	Description
SL_3DHINT_OFF	No hint is assigned, the implementation will perform rendering however it chooses.
SL_3DHINT_QUALITY_LOWEST, SL_3DHINT_QUALITY_LOW, SL_3DHINT_QUALITY_MEDIUM, SL_3DHINT_QUALITY_HIGH, SL_3DHINT_QUALITY_HIGHEST	These hinting values provide the rendering implementation with context as to the value the associated audio source may have to the listener. Values between these definitions may be used. The effect the hint has on rendering quality is implementation specific.

9.2.2 SL_AUDIOCODEC

```
#define SL_AUDIOCODEC_PCM                  ((SLuint32) 0x00000001)
#define SL_AUDIOCODEC_MP3                  ((SLuint32) 0x00000002)
#define SL_AUDIOCODEC_AMR                  ((SLuint32) 0x00000003)
#define SL_AUDIOCODEC_AMRWB                ((SLuint32) 0x00000004)
#define SL_AUDIOCODEC_AMRWBPLUS            ((SLuint32) 0x00000005)
#define SL_AUDIOCODEC_AAC                  ((SLuint32) 0x00000006)
#define SL_AUDIOCODEC_WMA                  ((SLuint32) 0x00000007)
#define SL_AUDIOCODEC_REAL                 ((SLuint32) 0x00000008)
#define SL_AUDIOCODEC_VORBIS               ((SLuint32) 0x00000009)
```

These macros are used for setting the audio encoding type.

Value	Description
SL_AUDIOCODEC_PCM	PCM audio data.
SL_AUDIOCODEC_MP3	MPEG Layer III encoder.
SL_AUDIOCODEC_AMR	Adaptive Multi-Rate (AMR) speech encoder.
SL_AUDIOCODEC_AMRWB	Adaptive Multi-Rate Wideband (AMR-WB) speech encoder.
SL_AUDIOCODEC_AMRWBPLUS	Adaptive Multi-Rate Wideband Extended (AMR-WB+) speech

Value	Description
	encoder.
SL_AUDIOCODEC_AAC	MPEG4 Advanced Audio Coding.
SL_AUDIOCODEC_WMA	Windows Media Audio.
SL_AUDIOCODEC_REAL	Real Audio.
SL_AUDIOCODEC_VORBIS	Vorbis Audio.

9.2.3 SL_AUDIOPROFILE and SL_AUDIOMODE

```
#define SL_AUDIOSTREAMFORMAT_UNDEFINED ((SLuint32) 0x00000000)
```

Value	Description
SL_AUDIOSTREAMFORMAT_UNDEFINED	The codec does not have a stream format.

PCM Profiles and Modes

```
#define SL_AUDIOPROFILE_PCM                  ((SLuint32) 0x00000001)
```

The macros are used for defining the PCM audio profiles.

Value	Description
SL_AUDIOPROFILE_PCM	Default Profile for PCM encoded Audio

MP3 Profiles and Modes

```
#define SL_AUDIOPROFILE_MPEG1_L3             ((SLuint32) 0x00000001)
#define SL_AUDIOPROFILE_MPEG2_L3             ((SLuint32) 0x00000002)
#define SL_AUDIOPROFILE_MPEG25_L3            ((SLuint32) 0x00000003)

#define SL_AUDIOCHANMODE_MP3_MONO            ((SLuint32) 0x00000001)
#define SL_AUDIOCHANMODE_MP3_STEREO          ((SLuint32) 0x00000002)
#define SL_AUDIOCHANMODE_MP3_JOINTSTEREO     ((SLuint32) 0x00000003)
#define SL_AUDIOCHANMODE_MP3_DUAL            ((SLuint32) 0x00000004)
```

The macros are used for defining the MP3 audio profiles and modes.

Value	Description
SL_AUDIOPROFILE_MPEG1_L3	MPEG-1 Layer III.
SL_AUDIOPROFILE_MPEG2_L3	MPEG-2 Layer III.
SL_AUDIOPROFILE_MPEG25_L3	MPEG-2.5 Layer III.
SL_AUDIOCHANMODE_MP3_MONO	MP3 Mono mode.
SL_AUDIOCHANMODE_MP3_STEREO	MP3 Stereo Mode.
SL_AUDIOCHANMODE_MP3_JOINTSTEREO	MP3 Joint Stereo mode.
SL_AUDIOCHANMODE_MP3_DUAL	MP3 Dual Stereo mode.

AMR Profiles and Modes

```
#define SL_AUDIOPROFILE_AMR                  ((SLuint32) 0x00000001)

#define SL_AUDIOSTREAMFORMAT_CONFORMANCE     ((SLuint32) 0x00000001)
#define SL_AUDIOSTREAMFORMAT_IF1             ((SLuint32) 0x00000002)
#define SL_AUDIOSTREAMFORMAT_IF2             ((SLuint32) 0x00000003)
#define SL_AUDIOSTREAMFORMAT_FSF             ((SLuint32) 0x00000004)
#define SL_AUDIOSTREAMFORMAT_RTPPAYLOAD      ((SLuint32) 0x00000005)
#define SL_AUDIOSTREAMFORMAT_ITU             ((SLuint32) 0x00000006)
```

The macros are used for defining the AMR audio profiles and modes.

Value	Description
SL_AUDIOPROFILE_AMR	Adaptive Multi-Rate audio codec.
SL_AUDIOSTREAMFORMAT_CONFORMANCE	Standard test-sequence format.
SL_AUDIOSTREAMFORMAT_IF1	Interface format 1.
SL_AUDIOSTREAMFORMAT_IF2	Interface format 2.
SL_AUDIOSTREAMFORMAT_FSF	File Storage format.
SL_AUDIOSTREAMFORMAT_RTPPAYLOAD	RTP payload format.
SL_AUDIOSTREAMFORMAT_ITU	ITU frame format.

AMR-WB Profiles and Modes

```
#define SL_AUDIOPROFILE_AMRWB                ((SLuint32) 0x00000001)
```

The macros are used for defining the AMR-WB audio profiles.

Value	Description
SL_AUDIOPROFILE_AMRWB	Adaptive Multi-Rate - Wideband.

AMR-WB+ Profiles and Modes

```
#define SL_AUDIOPROFILE_AMRWBPLUS            ((SLuint32) 0x00000001)
```

The macros are used for defining the AMR-WB+ audio profiles.

Value	Description
SL_AUDIOPROFILE_AMRWBPLUS	Extended Adaptive Multi-Rate – Wideband.

AAC Profiles and Modes

```
#define SL_AUDIOPROFILE_AAC_AAC              ((SLuint32) 0x00000001)

#define SL_AUDIOMODE_AAC_MAIN                ((SLuint32) 0x00000001)
#define SL_AUDIOMODE_AAC_LC                  ((SLuint32) 0x00000002)
#define SL_AUDIOMODE_AAC_SSR                 ((SLuint32) 0x00000003)
#define SL_AUDIOMODE_AAC_LTP                 ((SLuint32) 0x00000004)
#define SL_AUDIOMODE_AAC_HE                  ((SLuint32) 0x00000005)
#define SL_AUDIOMODE_AAC_SCALABLE            ((SLuint32) 0x00000006)
#define SL_AUDIOMODE_AAC_ERLC                ((SLuint32) 0x00000007)
#define SL_AUDIOMODE_AAC_LD                  ((SLuint32) 0x00000008)
#define SL_AUDIOMODE_AAC_HE_PS               ((SLuint32) 0x00000009)
#define SL_AUDIOMODE_AAC_HE_MPS              ((SLuint32) 0x0000000A)

#define SL_AUDIOSTREAMFORMAT_MP2ADTS         ((SLuint32) 0x00000001)
#define SL_AUDIOSTREAMFORMAT_MP4ADTS         ((SLuint32) 0x00000002)
#define SL_AUDIOSTREAMFORMAT_MP4LOAS         ((SLuint32) 0x00000003)
#define SL_AUDIOSTREAMFORMAT_MP4LATM         ((SLuint32) 0x00000004)
#define SL_AUDIOSTREAMFORMAT_ADIF            ((SLuint32) 0x00000005)
#define SL_AUDIOSTREAMFORMAT_MP4FF           ((SLuint32) 0x00000006)
#define SL_AUDIOSTREAMFORMAT_RAW             ((SLuint32) 0x00000007)
```

The macros are used for defining the AAC audio profiles and modes.

Value	Description
SL_AUDIOPROFILE_AAC_AAC	Advanced Audio Coding.
SL_AUDIOMODE_AAC_MAIN	AAC Main Profile.
SL_AUDIOMODE_AAC_LC	AAC Low Complexity.
SL_AUDIOMODE_AAC_SSR	AAC Scalable Sample Rate.
SL_AUDIOMODE_AAC_LTP	AAC Long Term Prediction.
SL_AUDIOMODE_AAC_HE	AAC High Efficiency.
SL_AUDIOMODE_AAC_SCALABLE	AAC Scalable.
SL_AUDIOMODE_AAC_ERLC	AAC Error Resilient LC.
SL_AUDIOMODE_AAC_LD	AAC Low Delay.

Value	Description
SL_AUDIOMODE_AAC_HE_PS	AAC High Efficiency with Parametric Stereo Coding.
SL_AUDIOMODE_AAC_HE_MPS	AAC High Efficiency with MPEG Surround Coding.
SL_AUDIOSTREAMFORMAT_MP2ADTS	MPEG-2 AAC Audio Data Transport Stream format.
SL_AUDIOSTREAMFORMAT_MP4ADTS	MPEG-4 AAC Audio Data Transport Stream format.
SL_AUDIOSTREAMFORMAT_MP4LOAS	Low Overhead Audio Stream format.
SL_AUDIOSTREAMFORMAT_MP4LATM	Low Overhead Audio Transport Multiplex.
SL_AUDIOSTREAMFORMAT_ADIF	Audio Data Interchange Format.
SL_AUDIOSTREAMFORMAT_MP4FF	AAC inside MPEG-4/ISO File Format.
SL_AUDIOSTREAMFORMAT_RAW	AAC Raw Format (access units).

Windows Media Audio Profiles and Modes

```
#define SL_AUDIOPROFILE_WMA7          ((SLuint32) 0x00000001)
#define SL_AUDIOPROFILE_WMA8          ((SLuint32) 0x00000002)
#define SL_AUDIOPROFILE_WMA9          ((SLuint32) 0x00000003)
#define SL_AUDIOPROFILE_WMA10         ((SLuint32) 0x00000004)

#define SL_AUDIOMODE_WMA_LEVEL1       ((SLuint32) 0x00000001)
#define SL_AUDIOMODE_WMA_LEVEL2       ((SLuint32) 0x00000002)
#define SL_AUDIOMODE_WMA_LEVEL3       ((SLuint32) 0x00000003)
#define SL_AUDIOMODE_WMA_LEVEL4       ((SLuint32) 0x00000004)
#define SL_AUDIOMODE_WMAPRO_LEVELM0   ((SLuint32) 0x00000005)
#define SL_AUDIOMODE_WMAPRO_LEVELM1   ((SLuint32) 0x00000006)
#define SL_AUDIOMODE_WMAPRO_LEVELM2   ((SLuint32) 0x00000007)
#define SL_AUDIOMODE_WMAPRO_LEVELM3   ((SLuint32) 0x00000008)
```

The macros are used for defining the WMA audio profiles and modes.

Value	Description
SL_AUDIOPROFILE_WMA7	Windows Media Audio Encoder V7.
SL_AUDIOPROFILE_WMA8	Windows Media Audio Encoder V8.
SL_AUDIOPROFILE_WMA9	Windows Media Audio Encoder V9.
SL_AUDIOPROFILE_WMA10	Windows Media Audio Encoder V10.
SL_AUDIOMODE_WMA_LEVEL1	WMA Level 1.
SL_AUDIOMODE_WMA_LEVEL2	WMA Level 2.
SL_AUDIOMODE_WMA_LEVEL3	WMA Level 3.
SL_AUDIOMODE_WMA_LEVEL3	WMA Level 4.
SL_AUDIOMODE_WMAPRO_LEVELM0	WMA Pro Level M0.

Value	Description
SL_AUDIOMODE_WMAPRO_LEVELM1	WMA Pro Level M1.
SL_AUDIOMODE_WMAPRO_LEVELM2	WMA Pro Level M2.
SL_AUDIOMODE_WMAPRO_LEVELM3	WMA Pro Level M3.

RealAudio Profiles and Levels

```
#define SL_AUDIOPROFILE_REALAUDIO            ((SLuint32) 0x00000001)

#define SL_AUDIOMODE_REALAUDIO_G2            ((SLuint32) 0x00000001)
#define SL_AUDIOMODE_REALAUDIO_8             ((SLuint32) 0x00000002)
#define SL_AUDIOMODE_REALAUDIO_10            ((SLuint32) 0x00000003)
#define SL_AUDIOMODE_REALAUDIO_SURROUND      ((SLuint32) 0x00000004)
```

The macros are used for defining the Real Audio audio profiles and modes.

Value	Description
SL_AUDIOPROFILE_REALAUDIO	RealAudio Encoder.
SL_AUDIOMODE_REALAUDIO_G2	RealAudio G2.
SL_AUDIOMODE_REALAUDIO_8	RealAudio 8.
SL_AUDIOMODE_REALAUDIO_10	RealAudio 10.
SL_AUDIOMODE_REALAUDIO_SURROUND	RealAudio Surround.

Vorbis Profiles and Levels

```
#define SL_AUDIOPROFILE_VORBIS               ((SLuint32) 0x00000001)

#define SL_AUDIOMODE_VORBIS                  ((SLuint32) 0x00000001)
```

The macros are used for defining the Vorbis audio profiles and modes.

Value	Description
SL_AUDIOPROFILE_VORBIS	Vorbis Encoder.
SL_AUDIOMODE_VORBIS	Default mode for Vorbis encoded audio.

9.2.4 SL_API

```
#define SL_API       <system dependent>
```

A system-dependent API function prototype declaration macro.

9.2.5 SLAPIENTRY

```
#define SLAPIENTRY   <system dependent>
```

A system-dependent API entry point macro. This may be used to indicate the required calling conventions for global functions.

9.2.6 SL_BOOLEAN

```
#define SL_BOOLEAN_FALSE   ((SLboolean) 0x00000000)
#define SL_BOOLEAN_TRUE    ((SLboolean) 0x00000001)
```

Canonical values for boolean type.

Value	Description
SL_BOOLEAN_FALSE	False value for SLboolean.
SL_BOOLEAN_TRUE	True value for SLboolean.

9.2.7 SL_BUFFERQUEUEEVENT

```
#define SL_BUFFERQUEUEEVENT_PROCESSED     ((SLuint32) 0x00000001)
#define SL_BUFFERQUEUEEVENT_UNREALIZED    ((SLuint32) 0x00000002)
#define SL_BUFFERQUEUEEVENT_CLEARED       ((SLuint32) 0x00000004)
#define SL_BUFFERQUEUEEVENT_STOPPED       ((SLuint32) 0x00000008)
#define SL_BUFFERQUEUEEVENT_ERROR         ((SLuint32) 0x00000010)
#define SL_BUFFERQUEUEEVENT_CONTENT_END   ((SLuint32) 0x00000020)
```

SL_BUFFERQUEUEEVENT represents the flags for a buffer queue event.

Value	Description
SL_BUFFERQUEUEEVENT_PROCESSED	All of the buffer data was processed by the player or recorder.
SL_BUFFERQUEUEEVENT_UNREALIZED	The buffer was dequeued due to the media object going into the SL_OBJECT_STATE_UNREALIZED state.
SL_BUFFERQUEUEEVENT_CLEARED	The buffer was dequeued by a call to SLBufferQueue::Clear().
SL_BUFFERQUEUEEVENT_STOPPED	The buffer was dequeued by the media object transitioning to the SL_PLAYSTATE_STOPPED or SL_RECORDSTATE_STOPPED.
SL_BUFFERQUEUEEVENT_ERROR	An error occurred with this buffer
SL_BUFFERQUEUEEVENT_CONTENT_END	This buffer is the last buffer containing content for either playback or recording.

9.2.8 SL_BYTEORDER

```
#define SL_BYTEORDER_BIGENDIAN          ((SLuint32) 0x00000001)
#define SL_BYTEORDER_LITTLEENDIAN       ((SLuint32) 0x00000002)
#define SL_BYTEORDER_NATIVE             <system dependent>
```

SL_BYTEORDER represents the byte order of a block of 16- or 32-bit data.

Value	Description
SL_BYTEORDER_BIGENDIAN	Big-endian data
SL_BYTEORDER_LITTLEENDIAN	Little-endian data
SL_BYTEORDER_NATIVE	Either big or little endian, based on system configuration via SL_BYTEORDER_NATIVEBIGENDIAN

9.2.9 SL_CHARACTERENCODING

```
#define SL_CHARACTERENCODING_UNKNOWN               ((SLuint32) 0x00000000)
#define SL_CHARACTERENCODING_BINARY                ((SLuint32) 0x00000001)
#define SL_CHARACTERENCODING_ASCII                 ((SLuint32) 0x00000002)
#define SL_CHARACTERENCODING_BIG5                  ((SLuint32) 0x00000003)
#define SL_CHARACTERENCODING_CODEPAGE1252          ((SLuint32) 0x00000004)
#define SL_CHARACTERENCODING_GB2312                ((SLuint32) 0x00000005)
#define SL_CHARACTERENCODING_HZGB2312              ((SLuint32) 0x00000006)
#define SL_CHARACTERENCODING_GB12345               ((SLuint32) 0x00000007)
#define SL_CHARACTERENCODING_GB18030               ((SLuint32) 0x00000008)
#define SL_CHARACTERENCODING_GBK                   ((SLuint32) 0x00000009)
#define SL_CHARACTERENCODING_IMAPUTF7              ((SLuint32) 0x0000000A)
#define SL_CHARACTERENCODING_ISO2022JP             ((SLuint32) 0x0000000B)
#define SL_CHARACTERENCODING_ISO2022JP1            ((SLuint32) 0x0000000B)
#define SL_CHARACTERENCODING_ISO88591              ((SLuint32) 0x0000000C)
#define SL_CHARACTERENCODING_ISO885910             ((SLuint32) 0x0000000D)
#define SL_CHARACTERENCODING_ISO885913             ((SLuint32) 0x0000000E)
#define SL_CHARACTERENCODING_ISO885914             ((SLuint32) 0x0000000F)
#define SL_CHARACTERENCODING_ISO885915             ((SLuint32) 0x00000010)
#define SL_CHARACTERENCODING_ISO88592              ((SLuint32) 0x00000011)
#define SL_CHARACTERENCODING_ISO88593              ((SLuint32) 0x00000012)
#define SL_CHARACTERENCODING_ISO88594              ((SLuint32) 0x00000013)
#define SL_CHARACTERENCODING_ISO88595              ((SLuint32) 0x00000014)
#define SL_CHARACTERENCODING_ISO88596              ((SLuint32) 0x00000015)
#define SL_CHARACTERENCODING_ISO88597              ((SLuint32) 0x00000016)
#define SL_CHARACTERENCODING_ISO88598              ((SLuint32) 0x00000017)
#define SL_CHARACTERENCODING_ISO88599              ((SLuint32) 0x00000018)
#define SL_CHARACTERENCODING_ISOEUCJP              ((SLuint32) 0x00000019)
#define SL_CHARACTERENCODING_SHIFTJIS              ((SLuint32) 0x0000001A)
#define SL_CHARACTERENCODING_SMS7BIT               ((SLuint32) 0x0000001B)
#define SL_CHARACTERENCODING_UTF7                  ((SLuint32) 0x0000001C)
#define SL_CHARACTERENCODING_UTF8                  ((SLuint32) 0x0000001D)
#define SL_CHARACTERENCODING_JAVACONFORMANTUTF8    ((SLuint32) 0x0000001E)
#define SL_CHARACTERENCODING_UTF16BE               ((SLuint32) 0x0000001F)
#define SL_CHARACTERENCODING_UTF16LE               ((SLuint32) 0x00000020)
```

SL_CHARACTERENCODING represents a character encoding for metadata keys and values.

Value	Description
SL_CHARACTERENCODING_UNKNOWN	Unknown character encoding.
SL_CHARACTERENCODING_BINARY	Binary data.
SL_CHARACTERENCODING_ASCII	ASCII.
SL_CHARACTERENCODING_BIG5	Big 5.
SL_CHARACTERENCODING_CODEPAGE1252	Microsoft Code Page 1252.
SL_CHARACTERENCODING_GB2312	GB 2312 (Chinese).
SL_CHARACTERENCODING_HZGB2312	HZ GB 2312 (Chinese).
SL_CHARACTERENCODING_GB12345	GB 12345 (Chinese).
SL_CHARACTERENCODING_GB18030	GB 18030 (Chinese).
SL_CHARACTERENCODING_GBK	GBK (CP936) (Chinese).
SL_CHARACTERENCODING_ISO2022JP	ISO-2022-JP (Japanese).
SL_CHARACTERENCODING_ISO2022JP1	ISO-2022-JP-1 (Japanese).
SL_CHARACTERENCODING_ISO88591	ISO-8859-1 (Latin-1).
SL_CHARACTERENCODING_ISO88592	ISO-8859-1 (Latin-2).
SL_CHARACTERENCODING_ISO88593	ISO-8859-1 (Latin-3).
SL_CHARACTERENCODING_ISO88594	ISO-8859-1 (Latin-4).
SL_CHARACTERENCODING_ISO88595	ISO-8859-1 (Latin/Cyrillic).
SL_CHARACTERENCODING_ISO88596	ISO-8859-1 (Latin/Arabic).
SL_CHARACTERENCODING_ISO88597	ISO-8859-1 (Latin/Greek).
SL_CHARACTERENCODING_ISO88598	ISO-8859-1 (Latin/Hebrew).
SL_CHARACTERENCODING_ISO88599	ISO-8859-1 (Latin-5).
SL_CHARACTERENCODING_ISO885910	ISO-8859-1 (Latin-6).
SL_CHARACTERENCODING_ISO885913	ISO-8859-1 (Latin-7).
SL_CHARACTERENCODING_ISO885914	ISO-8859-1 (Latin-8).
SL_CHARACTERENCODING_ISO885915	ISO-8859-1 (Latin-9).
SL_CHARACTERENCODING_ISOEUCJP	ISO EUC-JP.
SL_CHARACTERENCODING_SHIFTJIS	Shift-JIS (Japanese).
SL_CHARACTERENCODING_SMS7BIT	SMS 7-bit.
SL_CHARACTERENCODING_UTF7	Unicode UTF-7.
SL_CHARACTERENCODING_IMAPUTF7	Unicode UTF-7 per IETF RFC 2060.
SL_CHARACTERENCODING_UTF8	Unicode UTF-8.
SL_CHARACTERENCODING_JAVACONFORMANTUTF8	Unicode UTF-8 (Java Conformant).

Value	Description
SL_CHARACTERENCODING_UTF16BE	Unicode UTF-16 (Big Endian).
SL_CHARACTERENCODING_UTF16LE	Unicode UTF-16 (Little Endian).

9.2.10 SL_CONTAINERTYPE

```
#define SL_CONTAINERTYPE_UNSPECIFIED   ((SLuint32) 0x00000001)
#define SL_CONTAINERTYPE_RAW           ((SLuint32) 0x00000002)
#define SL_CONTAINERTYPE_ASF           ((SLuint32) 0x00000003)
#define SL_CONTAINERTYPE_AVI           ((SLuint32) 0x00000004)
#define SL_CONTAINERTYPE_BMP           ((SLuint32) 0x00000005)
#define SL_CONTAINERTYPE_JPG           ((SLuint32) 0x00000006)
#define SL_CONTAINERTYPE_JPG2000       ((SLuint32) 0x00000007)
#define SL_CONTAINERTYPE_M4A           ((SLuint32) 0x00000008)
#define SL_CONTAINERTYPE_MP3           ((SLuint32) 0x00000009)
#define SL_CONTAINERTYPE_MP4           ((SLuint32) 0x0000000A)
#define SL_CONTAINERTYPE_MPEG_ES       ((SLuint32) 0x0000000B)
#define SL_CONTAINERTYPE_MPEG_PS       ((SLuint32) 0x0000000C)
#define SL_CONTAINERTYPE_MPEG_TS       ((SLuint32) 0x0000000D)
#define SL_CONTAINERTYPE_QT            ((SLuint32) 0x0000000E)
#define SL_CONTAINERTYPE_WAV           ((SLuint32) 0x0000000F)
#define SL_CONTAINERTYPE_XMF_0         ((SLuint32) 0x00000010)
#define SL_CONTAINERTYPE_XMF_1         ((SLuint32) 0x00000011)
#define SL_CONTAINERTYPE_XMF_2         ((SLuint32) 0x00000012)
#define SL_CONTAINERTYPE_XMF_3         ((SLuint32) 0x00000013)
#define SL_CONTAINERTYPE_XMF_GENERIC   ((SLuint32) 0x00000014)
#define SL_CONTAINERTYPE_AMR           ((SLuint32) 0x00000015)
#define SL_CONTAINERTYPE_AAC           ((SLuint32) 0x00000016)
#define SL_CONTAINERTYPE_3GPP          ((SLuint32) 0x00000017)
#define SL_CONTAINERTYPE_3GA           ((SLuint32) 0x00000018)
#define SL_CONTAINERTYPE_RM            ((SLuint32) 0x00000019)
#define SL_CONTAINERTYPE_DMF           ((SLuint32) 0x0000001A)
#define SL_CONTAINERTYPE_SMF           ((SLuint32) 0x0000001B)
#define SL_CONTAINERTYPE_MOBILE_DLS    ((SLuint32) 0x0000001C)
#define SL_CONTAINERTYPE_OGG           ((SLuint32) 0x0000001D)
```

SL_CONTAINERTYPE represents the container type of the data source or sink.

Value	Description
SL_CONTAINERTYPE_UNSPECIFIED	The container type is not specified.
SL_CONTAINERTYPE_RAW	There is no container. Content is in raw form.
SL_CONTAINERTYPE_ASF	The container type is ASF.
SL_CONTAINERTYPE_AVI	The container type is AVI.
SL_CONTAINERTYPE_BMP	The container type is BMP.

Value	Description
SL_CONTAINERTYPE_JPG	The container type is JPEG.
SL_CONTAINERTYPE_JPG2000	The container type is JPEG 2000.
SL_CONTAINERTYPE_M4A	The container type is M4A.
SL_CONTAINERTYPE_MP3	The container type is MP3.
SL_CONTAINERTYPE_MP4	The container type is MP4.
SL_CONTAINERTYPE_MPEG_ES	The container type is MPEG Elementary Stream.
SL_CONTAINERTYPE_MPEG_PS	The container type is MPEG Program Stream.
SL_CONTAINERTYPE_MPEG_TS	The container type is MPEG Transport Stream.
SL_CONTAINERTYPE_QT	The container type is QuickTime.
SL_CONTAINERTYPE_WAV	The container type is WAV.
SL_CONTAINERTYPE_XMF_0	The container type is XMF Type 0.
SL_CONTAINERTYPE_XMF_1	The container type is XMF Type 1.
SL_CONTAINERTYPE_XMF_2	The container type is Mobile XMF (XMF Type 2).
SL_CONTAINERTYPE_XMF_3	The container type is Mobile XMF with Audio Clips (XMF Type 3).
SL_CONTAINERTYPE_XMF_GENERIC	The container type is the XMF Meta File Format (no particular XMF File Type)
SL_CONTAINERTYPE_AMR	This container type is the file storage format variant of AMR (the magic number in the header can be used to disambiguate between AMR-NB and AMR-WB).
SL_CONTAINERTYPE_AAC	This container type is for ADIF and ADTS variants of AAC. This refers to AAC in .aac files.
SL_CONTAINERTYPE_3GPP	The container type is 3GPP.
SL_CONTAINERTYPE_3GA	This container type is an audio-only variant of the 3GPP format, mainly used in 3G phones.
SL_CONTAINERTYPE_RM	This container type is Real Media.
SL_CONTAINERTYPE_DMF	This container type is Divx media format.
SL_CONTAINERTYPE_SMF	This container type is a standard MIDI file (SMF) [SP-MIDI].
SL_CONTAINERTYPE_MOBILE_DLS	This container type is a Mobile DLS file [mDLS].
SL_CONTAINERTYPE_OGG	This container type is OGG.

9.2.11 SL_DATAFORMAT

```
#define SL_DATAFORMAT_MIME       ((SLuint32) 0x00000001)
#define SL_DATAFORMAT_PCM        ((SLuint32) 0x00000002)
#define SL_DATAFORMAT_RESERVED3  ((SLuint32) 0x00000003)
#define SL_DATAFORMAT_PCM_EX     ((SLuint32) 0x00000004)
```

These values represent the possible data locators.

Value	Description
SL_DATAFORMAT_MIME	Data format is specified as a MIME type.
SL_DATAFORMAT_PCM	Data format is PCM. **(Deprecated)**
SL_DATAFORMAT_RESERVED3	Reserved value.
SL_DATAFORMAT_PCM_EX	Data format is PCM_EX.

9.2.12 SL_DATALOCATOR

```
#define SL_DATALOCATOR_NULL            ((SLuint32) 0x00000000)
#define SL_DATALOCATOR_URI             ((SLuint32) 0x00000001)
#define SL_DATALOCATOR_ADDRESS         ((SLuint32) 0x00000002)
#define SL_DATALOCATOR_IODEVICE        ((SLuint32) 0x00000003)
#define SL_DATALOCATOR_OUTPUTMIX       ((SLuint32) 0x00000004)
#define SL_DATALOCATOR_RESERVED5       ((SLuint32) 0x00000005)
#define SL_DATALOCATOR_BUFFERQUEUE     ((SLuint32) 0x00000006)
#define SL_DATALOCATOR_MIDIBUFFERQUEUE ((SLuint32) 0x00000007)
#define SL_DATALOCATOR_MEDIAOBJECT     ((SLuint32) 0x00000008)
#define SL_DATALOCATOR_CONTENTPIPE     ((SLuint32) 0x00000009)
```

These values represent the possible data locators.

Value	Description
SL_DATALOCATOR_NULL	No data will be generated or consumed.
SL_DATALOCATOR_URI	Data resides at the specified URI.
SL_DATALOCATOR_ADDRESS	Data is stored at the specified memory-mapped address.
SL_DATALOCATOR_IODEVICE	Data will be generated or consumed by the specified IO device. Note: for audio output use the output mix.
SL_DATALOCATOR_OUTPUTMIX	Data will be consumed by the specified audio output mix.
SL_DATALOCATOR_RESERVED5	Reserved value.
SL_DATALOCATOR_BUFFERQUEUE	Identifier for an SLDataLocator_BufferQueue.

Value	Description
SL_DATALOCATOR_MIDIBUFFERQUEUE	Identifier for an SLDataLocator_MIDIBufferQueue.
SL_DATALOCATOR_MEDIAOBJECT	Data will be generated or consumed by a media object.
SL_DATALOCATOR_CONTENTPIPE	Identifier for an SLDataLocator_ContentPipe

9.2.13 SL_DEFAULTDEVICEID

```
#define SL_DEFAULTDEVICEID_AUDIOINPUT  ((SLuint32) 0xFFFFFFFF)
#define SL_DEFAULTDEVICEID_AUDIOOUTPUT ((SLuint32) 0xFFFFFFFE)
#define SL_DEFAULTDEVICEID_LED         ((SLuint32) 0xFFFFFFFD)
#define SL_DEFAULTDEVICEID_VIBRA       ((SLuint32) 0xFFFFFFFC)
#define SL_DEFAULTDEVICEID_RESERVED1   ((SLuint32) 0xFFFFFFFB)
```

This macro may be used with any method that manipulates device IDs.

Value	Description
SL_DEFAULTDEVICEID_AUDIOINPUT	Identifier denoting the set of input devices from which the implementation receives audio from by default.
SL_DEFAULTDEVICEID_AUDIOOUTPUT	Identifier denoting the set of output devices to which the implementation sends audio to by default.
SL_DEFAULTDEVICEID_LED	Identifier denoting default LED array device.
SL_DEFAULTDEVICEID_VIBRA	Identifier denoting default vibra device.
SL_DEFAULTDEVICEID_RESERVED1	Reserved value.

9.2.14 SL_DEVICECONNECTION

```
#define SL_DEVCONNECTION_INTEGRATED         ((SLint16) 0x0001)
#define SL_DEVCONNECTION_ATTACHED_WIRED     ((SLint16) 0x0100)
#define SL_DEVCONNECTION_ATTACHED_WIRELESS  ((SLint16) 0x0200)
#define SL_DEVCONNECTION_NETWORK            ((SLint16) 0x0400)
```

These macros list the various types of I/O device connections possible. These connections are mutually exclusive for a given I/O device.

Value	Description
SL_DEVCONNECTION_INTEGRATED	I/O device is integrated onto the system (for example, mobile phone and, music player).

Value	**Description**
SL_DEVCONNECTION_ATTACHED_WIRED	I/O device is connected to the system via a wired connection. Additional macros might be added if more granularity is needed for each wired connection (such as USB, proprietary).
SL_DEVCONNECTION_ATTACHED_WIRELESS	I/O device is connected to the system via a wireless connection. Additional macros might be added if more granularity is needed for each wireless connection (such as Bluetooth).
SL_DEVCONNECTION_NETWORK	I/O device is connected to the system via *some* kind of network connection (either wired or wireless). This is different from the above connections (such as Bluetooth headset or wired accessory) in the sense that this connection could be to a remote device that could be quite distant geographically (unlike a Bluetooth headset or a wired headset that are in close proximity to the system). Also, a network connection implies going through some kind of network routing infrastructure that is not covered by the attached macros above. A Bluetooth headset or a wired headset represents a peer-to-peer connection, whereas a network connection does not. Examples of such network audio I/O devices include remote content servers that feed audio input to the system or a remote media renderer that plays out audio from the system, transmitted to it across a network.

9.2.15 SL_DEVICELOCATION

```
#define SL_DEVLOCATION_HANDSET   ((SLuint16) 0x0001)
#define SL_DEVLOCATION_HEADSET   ((SLuint16) 0x0002)
#define SL_DEVLOCATION_CARKIT    ((SLuint16) 0x0003)
#define SL_DEVLOCATION_DOCK      ((SLuint16) 0x0004)
#define SL_DEVLOCATION_REMOTE    ((SLuint16) 0x0005)
```

These macros list the location of the I/O device.

Value	Description
SL_DEVLOCATION_HANDSET	I/O device is on the handset.
SL_DEVLOCATION_HEADSET	I/O device is on a headset.
SL_DEVLOCATION_CARKIT	I/O device is on a carkit.
SL_DEVLOCATION_DOCK	I/O device is on a dock.
SL_DEVLOCATION_REMOTE	I/O device is in a remote location, most likely connected via some kind of a network.

Although it might seem like SL_DEVLOCATION_REMOTE is redundant since it is currently used with only SL_DEVCONNECTION_NETWORK, it is needed since none of the other device location macros fit a device whose connection type is SL_DEVCONNECTION_NETWORK.

9.2.16 SL_DEVICESCOPE

```
#define SL_DEVSCOPE_UNKNOWN      ((SLuint16) 0x0001)
#define SL_DEVSCOPE_ENVIRONMENT  ((SLuint16) 0x0002)
#define SL_DEVSCOPE_USER         ((SLuint16) 0x0003)
```

These macros list the scope of the I/O device with respect to the end user. These macros help the application to make routing decisions based on the type of content (such as audio) being rendered. For example, telephony downlink will always default to a “user” audio output device unless specifically changed by the user.

Value	Description
SL_DEVSCOPE_UNKNOWN	I/O device can have either a user scope or an environment scope or an as-yet-undefined scope. Good examples of audio I/O devices with such a scope would be line-in and line-out jacks. It is difficult to tell what types of devices will be plugged into these jacks. I/O devices connected via a network connection also fall into this category.
SL_DEVSCOPE_ENVIRONMENT	I/O device allows environmental (public) input or playback of content (such as audio). For example, an integrated loudspeaker is an “environmental” audio output device, since audio rendered to it can be heard by multiple people. Similarly, a microphone that can accept audio from multiple people is an “environmental” audio input device.
SL_DEVSCOPE_USER	I/O device allows input from or playback of content (such as audio) to a single user. For example, an earpiece speaker is a single-user audio output device since audio rendered to it can be heard only by one person. Similarly, the integrated microphone on a mobile phone is a single-user input device – it accepts input from just one person.

9.2.17 SL_DYNAMIC_ITF

```
#define SL_DYNAMIC_ITF_EVENT_RUNTIME_ERROR \
         ((SLuint32) 0x00000001)
#define SL_DYNAMIC_ITF_EVENT_ASYNC_TERMINATION \
         ((SLuint32) 0x00000002)
#define SL_DYNAMIC_ITF_EVENT_RESOURCES_LOST \
         ((SLuint32) 0x00000003)
#define SL_DYNAMIC_ITF_EVENT_RESOURCES_LOST_PERMANENTLY \
         ((SLuint32) 0x00000004)
#define SL_DYNAMIC_ITF_EVENT_RESOURCES_AVAILABLE \
         ((SLuint32) 0x00000005)
```

These values are used for identifying events used for dynamic interface management.

Value	Description
SL_DYNAMIC_ITF_EVENT_RUNTIME_ERROR	Runtime error.
SL_DYNAMIC_ITF_EVENT_ASYNC_TERMINATION	An asynchronous operation has terminated.
SL_DYNAMIC_ITF_EVENT_RESOURCES_LOST	Resources have been stolen from the dynamically managed interface, causing it to become suspended.
SL_DYNAMIC_ITF_EVENT_RESOURCES_LOST_PERMANENTLY	Resources have been stolen from the dynamically managed interface, causing it to become unrecoverable.
SL_DYNAMIC_ITF_EVENT_RESOURCES_AVAILABLE	Resources have become available, which may enable the dynamically managed interface to resume.

9.2.18 SL_ENGINEOPTION

```
#define SL_ENGINEOPTION_THREADSAFE     ((SLuint32) 0x00000001)
#define SL_ENGINEOPTION_LOSSOFCONTROL  ((SLuint32) 0x00000002)
#define SL_ENGINEOPTION_MAJORVERSION   ((SLuint32) 0x00000003)
#define SL_ENGINEOPTION_MINORVERSION   ((SLuint32) 0x00000004)
#define SL_ENGINEOPTION_STEPVERSION    ((SLuint32) 0x00000005)
```

Engine object creation options (see section 6.1).

Value	Description
SL_ENGINEOPTION_THREADSAFE	Thread safe engine creation option used with SLEngineOption structure [see section 9.1.20]. If the data field of the SLEngineOption structure is set to SL_BOOLEAN_TRUE, the engine object is created in thread-safe mode. Otherwise the engine object is created in non-thread-safe mode [see section 4.1.1].
SL_ENGINEOPTION_LOSSOFCONTROL	Global loss-of-control setting used with SLEngineOption structure [see section 9.1.20]. If the data field of the SLEngineOption structure is set to SL_BOOLEAN_TRUE, the engine object allows loss-of-control notifications to occur on interfaces. Otherwise, none of the interfaces exhibit loss-of-control behavior. This flag defaults to SL_BOOLEAN_FALSE if it is not explicitly turned on during engine creation. This global setting is best suited for applications that are interested in coarse-grained loss-of-control functionality; either it is allowed for that instance of the engine object or it is not. See SLObjectItf for details on loss-of-control.

Value	Description
`SL_ENGINEOPTION_MAJORVERSION`	The API major version for the requested engine object. The `data` field of the `SLEngineOption` structure is set to the integer major version of the requested engine object. The default value is 1.
`SL_ENGINEOPTION_MINORVERSION`	The API minor version for the requested engine object. The `data` field of the `SLEngineOption` structure is set to the integer minor version of the requested engine object. The default value is 0.
`SL_ENGINEOPTION_STEPVERSION`	The API step version for the requested engine object. The `data` field of the `SLEngineOption` structure is set to the integer step version of the requested engine object. Because step versions are backwards compatible, a higher step version of the engine than requested may be returned. The default value is 0.

9.2.19 SL_EQUALIZER

```
#define SL_EQUALIZER_UNDEFINED   ((SLuint16) 0xFFFF)
```

This value is used when equalizer setting is not defined.

Value	Description
`SL_EQUALIZER_UNDEFINED`	The setting is not defined.

9.2.20 SL_I3DL2 Environmental Reverb Presets

```
#define SL_I3DL2_ENVIRONMENT_PRESET_DEFAULT \
   { SL_MILLIBEL_MIN,  0, 1000,  500, SL_MILLIBEL_MIN, 20,
SL_MILLIBEL_MIN, 40, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_GENERIC \
   { -1000, -100, 1490, 830, -2602,  7,  200, 11, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_PADDEDCELL \
   { -1000,-6000, 170, 100, -1204,  1,  207,  2, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_ROOM \
   { -1000, -454, 400, 830, -1646,  2,  53,  3, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_BATHROOM \
   { -1000,-1200, 1490, 540, -370,  7, 1030, 11, 1000, 600 }
#define SL_I3DL2_ENVIRONMENT_PRESET_LIVINGROOM \
   { -1000,-6000, 500, 100, -1376,  3, -1104,  4, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_STONEROOM \
   { -1000, -300, 2310, 640, -711, 12,  83, 17, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_AUDITORIUM \
   { -1000, -476, 4320, 590, -789, 20, -289, 30, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_CONCERTHALL \
   { -1000, -500, 3920, 700, -1230, 20,  -2, 29, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_CAVE \
   { -1000,  0, 2910, 1300, -602, 15, -302, 22, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_ARENA \
   { -1000, -698, 7240, 330, -1166, 20,  16, 30, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_HANGAR \
   { -1000,-1000, 10050, 230, -602, 20,  198, 30, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_CARPETEDHALLWAY \
   { -1000,-4000, 300, 100, -1831,  2, -1630, 30, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_HALLWAY \
   { -1000, -300, 1490, 590, -1219,  7,  441, 11, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_STONECORRIDOR \
   { -1000, -237, 2700, 790, -1214, 13,  395, 20, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_ALLEY \
   { -1000, -270, 1490, 860, -1204,  7,  -4, 11, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_FOREST \
   { -1000,-3300, 1490, 540, -2560, 162, -613, 88, 790,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_CITY \
   { -1000, -800, 1490, 670, -2273,  7, -2217, 11, 500,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_MOUNTAINS \
   { -1000,-2500, 1490, 210, -2780, 300, -2014, 100, 270,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_QUARRY \
   { -1000,-1000, 1490, 830, SL_MILLIBEL_MIN, 61,  500, 25, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_PLAIN \
   { -1000,-2000, 1490, 500, -2466, 179, -2514, 100, 210,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_PARKINGLOT \
   { -1000,  0, 1650, 1500, -1363,  8, -1153, 12, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_SEWERPIPE \
   { -1000,-1000, 2810, 140,  429, 14,  648, 21, 800, 600 }
#define SL_I3DL2_ENVIRONMENT_PRESET_UNDERWATER \
   { -1000,-4000, 1490, 100, -449,  7, 1700, 11, 1000,1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_SMALLROOM \
   { -1000,-600, 1100, 830, -400, 5, 500, 10, 1000, 1000 }
```

```
#define SL_I3DL2_ENVIRONMENT_PRESET_MEDIUMROOM \
   { -1000,-600, 1300, 830, -1000, 20, -200, 20, 1000, 1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_LARGEROOM \
   { -1000,-600, 1500, 830, -1600, 5, -1000, 40, 1000, 1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_MEDIUMHALL \
   { -1000,-600, 1800, 700, -1300, 15, -800, 30, 1000, 1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_LARGEHALL \
   { -1000,-600, 1800, 700, -2000, 30, -1400, 60, 1000, 1000 }
#define SL_I3DL2_ENVIRONMENT_PRESET_PLATE \
   { -1000,-200, 1300, 900, 0, 2, 0, 10, 1000, 750 }
```

These macros are pre-defined sets of properties that are equivalent to those defined in the I3DL2 [I3DL2] help headers. These can be used for filling in the `SLEnvironmentalReverbSettings` structure [see section 9.1.21].

Value	Description
SL_I3DL2_ENVIRONMENT_PRESET_DEFAULT	Default environment, with no reverb.
SL_I3DL2_ENVIRONMENT_PRESET_GENERIC	Generic environment.
SL_I3DL2_ENVIRONMENT_PRESET_PADDEDCELL	Padded cell environment.
SL_I3DL2_ENVIRONMENT_PRESET_ROOM	Room environment.
SL_I3DL2_ENVIRONMENT_PRESET_BATHROOM	Bathroom environment.
SL_I3DL2_ENVIRONMENT_PRESET_LIVINGROOM	Living room environment.
SL_I3DL2_ENVIRONMENT_PRESET_STONEROOM	Stone room environment.
SL_I3DL2_ENVIRONMENT_PRESET_AUDITORIUM	Auditorium environment.
SL_I3DL2_ENVIRONMENT_PRESET_CONCERTHALL	Concert hall environment.
SL_I3DL2_ENVIRONMENT_PRESET_CAVE	Cave environment.
SL_I3DL2_ENVIRONMENT_PRESET_ARENA	Arena environment.
SL_I3DL2_ENVIRONMENT_PRESET_HANGAR	Hangar environment.
SL_I3DL2_ENVIRONMENT_PRESET_CARPETEDHALLWAY	Carpeted hallway environment.
SL_I3DL2_ENVIRONMENT_PRESET_HALLWAY	Hallway environment.
SL_I3DL2_ENVIRONMENT_PRESET_STONECORRIDOR	Stone corridor environment.
SL_I3DL2_ENVIRONMENT_PRESET_ALLEY	Alley environment.
SL_I3DL2_ENVIRONMENT_PRESET_FOREST	Forest environment.
SL_I3DL2_ENVIRONMENT_PRESET_CITY	City environment.
SL_I3DL2_ENVIRONMENT_PRESET_MOUNTAINS	Mountains environment.
SL_I3DL2_ENVIRONMENT_PRESET_QUARRY	Quarry environment.
SL_I3DL2_ENVIRONMENT_PRESET_PLAIN	Plain environment.
SL_I3DL2_ENVIRONMENT_PRESET_PARKINGLOT	Parking lot environment.

Value	Description
SL_I3DL2_ENVIRONMENT_PRESET_SEWERPIPE	Sewer pipe environment.
SL_I3DL2_ENVIRONMENT_PRESET_UNDERWATER	Underwater environment.
SL_I3DL2_ENVIRONMENT_PRESET_SMALLROOM	Small room environment.
SL_I3DL2_ENVIRONMENT_PRESET_MEDIUMROOM	Medium room environment.
SL_I3DL2_ENVIRONMENT_PRESET_LARGEROOM	Large room environment.
SL_I3DL2_ENVIRONMENT_PRESET_MEDIUMHALL	Medium hall environment.
SL_I3DL2_ENVIRONMENT_PRESET_LARGEHALL	Large hall environment.
SL_I3DL2_ENVIRONMENT_PRESET_PLATE	Plate environment.

9.2.21 SL_IODEVICE

```
#define SL_IODEVICE_AUDIOINPUT   ((SLuint32) 0x00000001)
#define SL_IODEVICE_LEDARRAY     ((SLuint32) 0x00000002)
#define SL_IODEVICE_VIBRA        ((SLuint32) 0x00000003)
#define SL_IODEVICE_RESERVED4    ((SLuint32) 0x00000004)
#define SL_IODEVICE_RESERVED5    ((SLuint32) 0x00000005)
#define SL_IODEVICE_AUDIOOUTPUT  ((SLuint32) 0x00000006)
```

These macros are used when creating I/O device data sources and sinks.

Value	Description
SL_IODEVICE_AUDIOINPUT	Device for audio input such as microphone or line-in.
SL_IODEVICE_LEDARRAY	Device for LED arrays.
SL_IODEVICE_VIBRA	Device for vibrators.
SL_IODEVICE_RESERVED4	Reserved.
SL_IODEVICE_RESERVED5	Reserved.
SL_IODEVICE_AUDIOOUTPUT	Device for audio output.

9.2.22 SL_METADATA_FILTER

```
#define SL_METADATA_FILTER_KEY        ((SLuint8) 0x01)
#define SL_METADATA_FILTER_LANG       ((SLuint8) 0x02)
#define SL_METADATA_FILTER_ENCODING   ((SLuint8) 0x04)
```

Bit-masks for metadata filtering criteria.

Value	Description
SL_METADATA_FILTER_KEY	Enable filtering by key.

Value	Description
SL_METADATA_FILTER_LANG	Enable filtering by language / country code.
SL_METADATA_FILTER_ENCODING	Enable filtering by value encoding.

9.2.23 SL_METADATATRAVERSALMODE

```
#define SL_METADATATRAVERSALMODE_ALL   ((SLuint32) 0x00000001)
#define SL_METADATATRAVERSALMODE_NODE  ((SLuint32) 0x00000002)
```

SL_METADATATRAVERSALMODE represents a method of traversing metadata within a file.

Value	Description
SL_METADATATRAVERSALMODE_ALL	Search the file linearly without considering its logical organization.
SL_METADATATRAVERSALMODE_NODE	Search by individual nodes, boxes, chunks, etc. within a file. (This is the default mode, with the default active node being the root node.)

9.2.24 SL_MIDIMESSAGETYPE

```
#define SL_MIDIMESSAGETYPE_NOTE_ON_OFF       ((SLuint32) 0x00000001)
#define SL_MIDIMESSAGETYPE_POLY_PRESSURE     ((SLuint32) 0x00000002)
#define SL_MIDIMESSAGETYPE_CONTROL_CHANGE    ((SLuint32) 0x00000003)
#define SL_MIDIMESSAGETYPE_PROGRAM_CHANGE    ((SLuint32) 0x00000004)
#define SL_MIDIMESSAGETYPE_CHANNEL_PRESSURE  ((SLuint32) 0x00000005)
#define SL_MIDIMESSAGETYPE_PITCH_BEND        ((SLuint32) 0x00000006)
#define SL_MIDIMESSAGETYPE_SYSTEM_MESSAGE    ((SLuint32) 0x00000007)
```

SL_MIDIMESSAGETYPE is used for filtering MIDI messages

Value	Description
SL_MIDIMESSAGETYPE_NOTE_ON_OFF	Note On / Note Off messages (status bytes 8n / 9n)
SL_MIDIMESSAGETYPE_POLY_PRESSURE	Polyphonic key pressure / Aftertouch messages (status byte An)
SL_MIDIMESSAGETYPE_CONTROL_CHANGE	Control change messages (status byte Bn)
SL_MIDIMESSAGETYPE_PROGRAM_CHANGE	Program change messages (status byte Cn)
SL_MIDIMESSAGETYPE_CHANNEL_PRESSURE	Channel pressure / Aftertouch messages (status byte Dn)
SL_MIDIMESSAGETYPE_PITCH_BEND	Pitch bend change messages (status byte En)
SL_MIDIMESSAGETYPE_SYSTEM_MESSAGE	System messages, including System Exclusive, System Common, and System Real Time (status byte Fn)

9.2.25 SL_MILLIBEL

```
#define SL_MILLIBEL_MAX    ((SLmillibel) 0x7FFF)
#define SL_MILLIBEL_MIN    ((SLmillibel) (-SL_MILLIBEL_MAX-1))
```

Limit values for millibel units.

Value	Description
SL_MILLIBEL_MAX	Maximum volume level.
SL_MILLIBEL_MIN	Minimum volume level. This volume may be treated as silence in some implementations.

9.2.26 SL_MILLIHERTZ_MAX

```
#define SL_MILLIHERTZ_MAX  ((SLmilliHertz) 0xFFFFFFFF)
```

Limit value for milliHertz unit.

Value	Description
SL_MILLIHERTZ_MAX	A macro for representing the maximum possible frequency.

9.2.27 SL_MILLIMETER_MAX

```
#define SL_MILLIMETER_MAX  ((SLmillimeter) 0x7FFFFFFF)
```

Limit value for millimeter unit.

Value	Description
SL_MILLIMETER_MAX	A macro for representing the maximum possible positive distance.

9.2.28 SL_NODE_PARENT

```
#define SL_NODE_PARENT     ((SLuint32) 0xFFFFFFFF)
```

SL_NODE_PARENT is used by SLMetadataTraversalItf::SetActiveNode to set the current scope to the node’s parent.

Value	Description
SL_NODE_PARENT	Used for setting the active parent node.

9.2.29 SL_NODETYPE

```
#define SL_NODETYPE_UNSPECIFIED  ((SLuint32) 0x00000001)
#define SL_NODETYPE_AUDIO        ((SLuint32) 0x00000002)
#define SL_NODETYPE_VIDEO        ((SLuint32) 0x00000003)
#define SL_NODETYPE_IMAGE        ((SLuint32) 0x00000004)
```

SL_NODETYPE represents the type of a node.

Value	Description
SL_NODETYPE_UNSPECIFIED	Unspecified node type.
SL_NODETYPE_AUDIO	Audio node.
SL_NODETYPE_VIDEO	Video node.
SL_NODETYPE_IMAGE	Image node.

9.2.30 SL_OBJECT_EVENT

```
#define SL_OBJECT_EVENT_RUNTIME_ERROR            ((SLuint32) 0x00000001)
#define SL_OBJECT_EVENT_ASYNC_TERMINATION        ((SLuint32) 0x00000002)
#define SL_OBJECT_EVENT_RESOURCES_LOST           ((SLuint32) 0x00000003)
#define SL_OBJECT_EVENT_RESOURCES_AVAILABLE      ((SLuint32) 0x00000004)
#define SL_OBJECT_EVENT_ITF_CONTROL_TAKEN        ((SLuint32) 0x00000005)
#define SL_OBJECT_EVENT_ITF_CONTROL_RETURNED     ((SLuint32) 0x00000006)
#define SL_OBJECT_EVENT_ITF_PARAMETERS_CHANGED   ((SLuint32) 0x00000007)
```

The macros identify the various event notifications that an object may emit.

Value	**Description**
SL_OBJECT_EVENT_RUNTIME_ERROR	Runtime error.
SL_OBJECT_EVENT_ASYNC_TERMINATION	An asynchronous operation has terminated.
SL_OBJECT_EVENT_RESOURCES_LOST	Resources have been stolen from the object, causing it to become unrealized or suspended.
SL_OBJECT_EVENT_RESOURCES_AVAILABLE	Resources have become available, which may enable the object to recover.
SL_OBJECT_EVENT_ITF_CONTROL_TAKEN	An interface has lost control. This event cannot be followed by another SL_OBJECT_EVENT_ITF_CONTROL_TAKEN event (for the interface in question).
SL_OBJECT_EVENT_ITF_CONTROL_RETURNED	Control was returned to an interface. This event cannot be followed by another SL_OBJECT_EVENT_ITF_CONTROL_RETURNED event (for the interface in question).
SL_OBJECT_EVENT_ITF_PARAMETERS_CHANGED	Some of the parameters of the interface in question were changed by other entity. (If the application wants to know the new values, it should use getters.) This event can only occur (for the interface in question) between SL_OBJECT_EVENT_ITF_CONTROL_TAKEN and SL_OBJECT_EVENT_ITF_CONTROL_RETURNED events.

9.2.31 SL_OBJECT_STATE

```
#define SL_OBJECT_STATE_UNREALIZED     ((SLuint32) 0x00000001)
#define SL_OBJECT_STATE_REALIZED       ((SLuint32) 0x00000002)
#define SL_OBJECT_STATE_SUSPENDED      ((SLuint32) 0x00000003)
```

These macros are used to identify the object states.

Value	Description
SL_OBJECT_STATE_UNREALIZED	Unrealized state.
SL_OBJECT_STATE_REALIZED	Realized state.
SL_OBJECT_STATE_SUSPENDED	Suspended state.

9.2.32 SL_OBJECTID

```
#define SL_OBJECTID_ENGINE             ((SLuint32) 0x00001001)
#define SL_OBJECTID_LEDDEVICE          ((SLuint32) 0x00001002)
#define SL_OBJECTID_VIBRADEVICE        ((SLuint32) 0x00001003)
#define SL_OBJECTID_AUDIOPLAYER        ((SLuint32) 0x00001004)
#define SL_OBJECTID_AUDIORECORDER      ((SLuint32) 0x00001005)
#define SL_OBJECTID_MIDIPLAYER         ((SLuint32) 0x00001006)
#define SL_OBJECTID_LISTENER           ((SLuint32) 0x00001007)
#define SL_OBJECTID_3DGROUP            ((SLuint32) 0x00001008)
#define SL_OBJECTID_OUTPUTMIX          ((SLuint32) 0x00001009)
#define SL_OBJECTID_METADATAEXTRACTOR  ((SLuint32) 0x0000100A)
```

These macros are the object type identifiers use while querying for the supported interfaces.

Value	Description
SL_OBJECTID_ENGINE	Engine Object ID.
SL_OBJECTID_LEDDEVICE	LED Device Object ID.
SL_OBJECTID_VIBRADEVICE	Vibra Device Object ID.
SL_OBJECTID_AUDIOPLAYER	Audio Player Object ID.
SL_OBJECTID_MIDIPLAYER	MIDI Player Object ID.
SL_OBJECTID_LISTENER	Listener Object ID.
SL_OBJECTID_3DGROUP	3D Group Object ID.
SL_OBJECTID_OUTPUTMIX	Output Mix Object ID.
SL_OBJECTID_METADATAEXTRACTOR	Metadata Extractor Object ID.

9.2.33 SL_PCM_REPRESENTATION

```
#define SL_PCM_REPRESENTATION_SIGNED_INT        ((SLuint32) 0x00000001)
#define SL_PCM_REPRESENTATION_UNSIGNED_INT      ((SLuint32) 0x00000002)
#define SL_PCM_REPRESENTATION_FLOAT             ((SLuint32) 0x00000003)
```

SL_PCM_REPRESENTATION denotes the type of PCM data.

Value	Description
SL_PCM_REPRESENTATION_SIGNED_INT	Signed integer data.
SL_PCM_REPRESENTATION_UNSIGNED_INT	Unsigned integer data.
SL_PCM_REPRESENTATION_FLOAT	Floating-point data.

9.2.34 SL_PCMSAMPLEFORMAT

```
   #define SL_PCMSAMPLEFORMAT_FIXED_8     ((SLuint16) 0x0008)
   #define SL_PCMSAMPLEFORMAT_FIXED_16    ((SLuint16) 0x0010)
   #define SL_PCMSAMPLEFORMAT_FIXED_20    ((SLuint16) 0x0014)
   #define SL_PCMSAMPLEFORMAT_FIXED_24    ((SLuint16) 0x0018)
   #define SL_PCMSAMPLEFORMAT_FIXED_28    ((SLuint16) 0x001C)
   #define SL_PCMSAMPLEFORMAT_FIXED_32    ((SLuint16) 0x0020)
   #define SL_PCMSAMPLEFORMAT_FIXED_64    ((SLuint16) 0x0040)
```

These macros list the various sample formats that are possible on audio input and output devices.

Value	Description
SL_PCMSAMPLEFORMAT_FIXED_8	Fixed-point 8-bit samples in 8-bit container.
SL_PCMSAMPLEFORMAT_FIXED_16	Fixed-point 16-bit samples in 16 bit container.
SL_PCMSAMPLEFORMAT_FIXED_20	Fixed-point 20-bit samples in 32 bit container left-justified.
SL_PCMSAMPLEFORMAT_FIXED_24	Fixed-point 24-bit samples in 32 bit container left-justified.
SL_PCMSAMPLEFORMAT_FIXED_28	Fixed-point 28-bit samples in 32 bit container left-justified.
SL_PCMSAMPLEFORMAT_FIXED_32	Fixed-point 32-bit samples in 32 bit container left-justified.
SL_PCMSAMPLEFORMAT_FIXED_64	Fixed-point 64-bit samples in 64 bit container left-justified.

9.2.35 SL_PLAYEVENT

```
#define SL_PLAYEVENT_HEADATEND         ((SLuint32) 0x00000001)
#define SL_PLAYEVENT_HEADATMARKER      ((SLuint32) 0x00000002)
#define SL_PLAYEVENT_HEADATNEWPOS      ((SLuint32) 0x00000004)
#define SL_PLAYEVENT_HEADMOVING        ((SLuint32) 0x00000008)
#define SL_PLAYEVENT_HEADSTALLED       ((SLuint32) 0x00000010)
#define SL_PLAYEVENT_DURATIONUPDATED   ((SLuint32) 0x00000020)
```

These values represent the possible play events.

Value	Description
SL_PLAYEVENT_HEADATEND	Playback head is at the end of the current content and the player has paused.
SL_PLAYEVENT_HEADATMARKER	Playback head is at the specified marker position.
SL_PLAYEVENT_HEADATNEWPOS	Playback head is at a new position (period between notifications is specified in by application).
SL_PLAYEVENT_HEADMOVING	Playback head has begun to move.
SL_PLAYEVENT_HEADSTALLED	Playback head has temporarily stopped moving.
SL_PLAYEVENT_DURATIONUPDATED	The duration of the content has been updated.

9.2.36 SL_PLAYSTATE

```
#define SL_PLAYSTATE_STOPPED     ((SLuint32) 0x00000001)
#define SL_PLAYSTATE_PAUSED      ((SLuint32) 0x00000002)
#define SL_PLAYSTATE_PLAYING     ((SLuint32) 0x00000003)
```

These values represent the playback state of an object

Value	Description
SL_PLAYSTATE_STOPPED	Player is stopped. The playback head is forced to the beginning of the content and is not trying to move.
SL_PLAYSTATE_PAUSED	Player is paused. The playback head may be anywhere within the content but is not trying to move.
SL_PLAYSTATE_PLAYING	Player is playing. The playback head may be anywhere within the content and is trying to move.

9.2.37 SL_PREFETCHEVENT

```
#define SL_PREFETCHEVENT_STATUSCHANGE       ((SLuint32) 0x00000001)
#define SL_PREFETCHEVENT_FILLLEVELCHANGE    ((SLuint32) 0x00000002)
#define SL_PREFETCHEVENT_ERROR              ((SLuint32) 0x00000003)
#define SL_PREFETCHEVENT_ERROR_UNRECOVERABLE ((SLuint32) 0x00000004)
```

These values represent the possible prefetch related events.

Value	Description
SL_PREFETCHEVENT_STATUSCHANGE	Prefetch status has changed.
SL_PREFETCHEVENT_FILLLEVELCHANGE	Prefetch fill level has changed.
SL_PREFETCHEVENT_ERROR	An error occured during prefetching.
SL_PREFETCHEVENT_ERROR_UNRECOVERABLE	An error occured during prefetching, and prefetching cannot resume.

9.2.38 SL_PREFETCHSTATUS

```
#define SL_PREFETCHSTATUS_UNDERFLOW         ((SLuint32) 0x00000001)
#define SL_PREFETCHSTATUS_SUFFICIENTDATA    ((SLuint32) 0x00000002)
#define SL_PREFETCHSTATUS_OVERFLOW          ((SLuint32) 0x00000003)
```

These values represent the possible status of a player’s prefetching operation.

Value	Description
SL_PREFETCHSTATUS_UNDERFLOW	Playback is suffering due to data starvation.
SL_PREFETCHSTATUS_SUFFICIENTDATA	Playback is not suffering due to data starvation or spillover.
SL_PREFETCHSTATUS_OVERFLOW	Playback is suffering due to data spillover.

9.2.39 SL_PRIORITY

```
#define SL_PRIORITY_LOWEST       ((SLuint32) 0xFFFFFFFF)
#define SL_PRIORITY_VERYLOW      ((SLuint32) 0xE0000000)
#define SL_PRIORITY_LOW          ((SLuint32) 0xC0000000)
#define SL_PRIORITY_BELOWNORMAL  ((SLuint32) 0xA0000000)
#define SL_PRIORITY_NORMAL       ((SLuint32) 0x7FFFFFFF)
#define SL_PRIORITY_ABOVENORMAL  ((SLuint32) 0x60000000)
#define SL_PRIORITY_HIGH         ((SLuint32) 0x40000000)
#define SL_PRIORITY_VERYHIGH     ((SLuint32) 0x20000000)
#define SL_PRIORITY_HIGHEST      ((SLuint32) 0x00000000)
```

Convenient macros representing various different priority levels, for use with the `SetPriority` method.

Value	Description
SL_PRIORITY_LOWEST	The lowest specifiable priority.
SL_PRIORITY_VERYLOW	Very low priority.
SL_PRIORITY_LOW	Low priority.
SL_PRIORITY_BELOWNORMAL	Below normal priority.
SL_PRIORITY_NORMAL	Normal priority given to objects.
SL_PRIORITY_ABOVENORMAL	Above normal priority.
SL_PRIORITY_HIGH	High priority.
SL_PRIORITY_VERYHIGH	Very high priority.
SL_PRIORITY_HIGHEST	Highest specifiable priority.

9.2.40 SL_PROFILES

```
#define SL_PROFILES_PHONE  ((SLuint16) 0x0001)
#define SL_PROFILES_MUSIC  ((SLuint16) 0x0002)
#define SL_PROFILES_GAME   ((SLuint16) 0x0004)
```

These macros list the 3 profiles of the OpenSL ES API.

Value	Description
SL_PROFILES_PHONE	Phone profile of OpenSL ES (see section 2.3 for a detailed description of all three profiles)
SL_PROFILES_MUSIC	Music profile of OpenSL ES.
SL_PROFILES_GAME	Game profile of OpenSL ES.

9.2.41 SL_RATECONTROLMODE

```
#define SL_RATECONTROLMODE_CONSTANTBITRATE   ((SLuint32) 0x00000001)
#define SL_RATECONTROLMODE_VARIABLEBITRATE   ((SLuint32) 0x00000002)
```

These defines are used to set the rate control mode.

Value	Description
SL_RATECONTROLMODE_CONSTANTBITRATE	Constant bitrate mode.
SL_RATECONTROLMODE_VARIABLEBITRATE	Variable bitrate mode.

9.2.42 SL_RATEPROP

```
#define SL_RATEPROP_RESERVED1          ((SLuint32) 0x00000001)
#define SL_RATEPROP_RESERVED2          ((SLuint32) 0x00000002)
#define SL_RATEPROP_SILENTAUDIO        ((SLuint32) 0x00000100)
#define SL_RATEPROP_STAGGEREDAUDIO     ((SLuint32) 0x00000200)
#define SL_RATEPROP_NOPITCHCORAUDIO    ((SLuint32) 0x00000400)
#define SL_RATEPROP_PITCHCORAUDIO      ((SLuint32) 0x00000800)
```

These values represent the rate-related properties of an object.

Value	Description
SL_RATEPROP_RESERVED1	Reserved.
SL_RATEPROP_RESERVED2	Reserved.
SL_RATEPROP_SILENTAUDIO	Silences audio output. This property accommodates limitations of rewind and high speed fast-forward.
SL_RATEPROP_STAGGEREDAUDIO	Plays small chunks of audio at 1x forward, skipping segments of audio between chunks. The progression of the playback head between chunks obeys the direction and speed implied by the current rate. This property accommodates limitations of rewind and high speed fast forward.
SL_RATEPROP_NOPITCHCORAUDIO	Plays audio at the current rate, but without pitch correction.
SL_RATEPROP_PITCHCORAUDIO	Plays audio at the current rate, but with pitch correction.

9.2.43 SL_RECORDEVENT

```
#define SL_RECORDEVENT_HEADATLIMIT            ((SLuint32) 0x00000001)
#define SL_RECORDEVENT_HEADATMARKER           ((SLuint32) 0x00000002)
#define SL_RECORDEVENT_HEADATNEWPOS           ((SLuint32) 0x00000004)
#define SL_RECORDEVENT_HEADMOVING             ((SLuint32) 0x00000008)
#define SL_RECORDEVENT_HEADSTALLED            ((SLuint32) 0x00000010)
#define SL_RECORDEVENT_BUFFER_FULL            ((SLuint32) 0x00000020)
#define SL_RECORDEVENT_BUFFERQUEUE_STARVED    ((SLuint32) 0x00000040)
```

These values represent the possible record events.

Value	Description
SL_RECORDEVENT_HEADATLIMIT	Recording head is at the specified duration limit and the recorder has stopped.
SL_RECORDEVENT_HEADATMARKER	Recording head is at the specified marker position.

Value	Description
SL_RECORDEVENT_HEADATNEWPOS	Recording head is at a new position. (Period between notifications is specified by application.)
SL_RECORDEVENT_HEADMOVING	Recording head has begun to move.
SL_RECORDEVENT_HEADSTALLED	Recording head has temporarily stopped moving.
SL_RECORDEVENT_BUFFER_FULL	Recording has reached the end of the memory buffer (i.e. SLDataLocator_Address). When the recorder is unable to write any more data (for example, when the memory buffer it is writing to is full) the recorder transitions to the SL_RECORDSTATE_STOPPED state. This event will not be posted when recording to a file.
SL_RECORDEVENT_BUFFERQUEUE_STARVED	A lack of available buffers has caused a loss of recorded data.

9.2.44 SL_RECORDSTATE

```
#define SL_RECORDSTATE_STOPPED   ((SLuint32) 0x00000001)
#define SL_RECORDSTATE_PAUSED    ((SLuint32) 0x00000002)
#define SL_RECORDSTATE_RECORDING ((SLuint32) 0x00000003)
```

These values represent the recording state of an object.

Value	Description
SL_RECORDSTATE_STOPPED	Recorder is stopped. The destination is closed
SL_RECORDSTATE_PAUSED	Recorder is stopped. The destination is open but not receiving captured content.
SL_RECORDSTATE_RECORDING	Recorder is recording. The destination is open and receiving captured content.

9.2.45 SL_REVERBPRESET

```
#define SL_REVERBPRESET_NONE             ((Sluint16) 0x0000)
#define SL_REVERBPRESET_SMALLROOM        ((SLuint16) 0x0001)
#define SL_REVERBPRESET_MEDIUMROOM       ((SLuint16) 0x0002)
#define SL_REVERBPRESET_LARGEROOM        ((SLuint16) 0x0003)
#define SL_REVERBPRESET_MEDIUMHALL       ((Sluint16) 0x0004)
#define SL_REVERBPRESET_LARGEHALL        ((SLuint16) 0x0005)
#define SL_REVERBPRESET_PLATE            ((SLuint16) 0x0006)
```

These macros define the reverb presets supported by the `SLPresetReverbItf` interface. These presets are based on the music presets in I3DL2 guidelines [I3DL2].

Value	Description
SL_REVERBPRESET_NONE	No reverb of reflections.
SL_REVERBPRESET_SMALLROOM	Reverb preset representing a small room less than five meters in length.
SL_REVERBPRESET_MEDIUMROOM	Reverb preset representing a medium room with a length of ten meters or less.
SL_REVERBPRESET_LARGEROOM	Reverb preset representing a large-sized room suitable for live performances.
SL_REVERBPRESET_MEDIUMHALL	Reverb preset representing a medium-sized hall.
SL_REVERBPRESET_LARGEHALL	Reverb preset representing a large-sized hall suitable for a full orchestra.
SL_REVERBPRESET_PLATE	Reverb preset representing a synthesis of the traditional *plate* reverb.

9.2.46 SL_RESULT

```
#define SL_RESULT_SUCCESS                      ((SLuint32) 0x00000000)
#define SL_RESULT_PRECONDITIONS_VIOLATED       ((SLuint32) 0x00000001)
#define SL_RESULT_PARAMETER_INVALID            ((SLuint32) 0x00000002)
#define SL_RESULT_MEMORY_FAILURE               ((SLuint32) 0x00000003)
#define SL_RESULT_RESOURCE_ERROR               ((SLuint32) 0x00000004)
#define SL_RESULT_RESOURCE_LOST                ((SLuint32) 0x00000005)
#define SL_RESULT_IO_ERROR                     ((SLuint32) 0x00000006)
#define SL_RESULT_BUFFER_INSUFFICIENT          ((SLuint32) 0x00000007)
#define SL_RESULT_CONTENT_CORRUPTED            ((SLuint32) 0x00000008)
#define SL_RESULT_CONTENT_UNSUPPORTED          ((SLuint32) 0x00000009)
#define SL_RESULT_CONTENT_NOT_FOUND            ((SLuint32) 0x0000000A)
#define SL_RESULT_PERMISSION_DENIED            ((SLuint32) 0x0000000B)
#define SL_RESULT_FEATURE_UNSUPPORTED          ((SLuint32) 0x0000000C)
#define SL_RESULT_INTERNAL_ERROR               ((SLuint32) 0x0000000D)
#define SL_RESULT_UNKNOWN_ERROR                ((SLuint32) 0x0000000E)
#define SL_RESULT_OPERATION_ABORTED            ((SLuint32) 0x0000000F)
#define SL_RESULT_CONTROL_LOST                 ((SLuint32) 0x00000010)
#define SL_RESULT_READONLY                     ((SLuint32) 0x00000011)
```

```
#define SL_RESULT_ENGINEOPTION_UNSUPPORTED   ((SLuint32) 0x00000012)
#define SL_RESULT_SOURCE_SINK_INCOMPATIBLE   ((SLuint32) 0x00000013)
```

The `SL_RESULT` values are described.

Value	Description
SL_RESULT_SUCCESS	Success.
SL_RESULT_PRECONDITIONS_VIOLATED	Use of the method violates a pre-condition (not including invalid parameters). The pre-conditions are defined in the method specifications.
SL_RESULT_PARAMETER_INVALID	An invalid parameter has been detected. In case of parameters passed by pointer (such as the self-parameters) – if the pointer is corrupt, an implementation's behavior is undefined. However, it is recommended that implementations at least check for `NULL`-pointers.
SL_RESULT_MEMORY_FAILURE	The method was unable to allocate or release memory.
SL_RESULT_RESOURCE_ERROR	Operation failed due to a lack of resources (usually a result of object realization).
SL_RESULT_RESOURCE_LOST	Operation ignored, since object is in Unrealized or Suspended state.
SL_RESULT_IO_ERROR	Failure due to an I/O error (file or other I/O device).
SL_RESULT_BUFFER_INSUFFICIENT	One or more of the buffers passed to the method is too small to service the request.
SL_RESULT_CONTENT_CORRUPTED	Failure due to corrupted content (also applies for malformed MIDI messages sent programmatically).
SL_RESULT_CONTENT_UNSUPPORTED	Failure due to an unsupported content format (such as unsupported codec).
SL_RESULT_CONTENT_NOT_FOUND	Failed to retrieve content (for example, file not found).
SL_RESULT_PERMISSION_DENIED	Failure due to violation of DRM, user permissions, policies, etc.

Value	Description
SL_RESULT_FEATURE_UNSUPPORTED	Failure due to an unsupported feature. This can occur either when calling `GetInterface()` on an unsupported interface or when attempting to call a method not supported in an interface. This can also occur when an interface with an unknown ID is used (either during object creation or in a `GetInterface()` call). See Section 2.6.
SL_RESULT_INTERNAL_ERROR	Failure due to an (unrecoverable) internal error.
SL_RESULT_UNKNOWN_ERROR	Catch-all error, including system errors. Should never be returned when any of the above errors apply.
SL_RESULT_OPERATION_ABORTED	Operation was aborted as a result of a user request.
SL_RESULT_CONTROL_LOST	Another entity is now controlling the interface and it cannot be controlled by this application currently. `slObjectCallback` can be used for monitoring this behavior: this error code can only occur between SL_OBJECT_EVENT_ITF_CONTROL_TAKEN and SL_OBJECT_EVENT_ITF_CONTROL_RETURNED events.
SL_RESULT_SOURCE_SINK_INCOMPATIBLE	The source and sink being connected are incompatible (incompatibility could be due to format and/or locator type mismatch)..
SL_RESULT_READONLY	Operation failed because the parameter in question is Read-Only.
SL_RESULT_ENGINEOPTION_UNSUPPORTED	Failure due to an unsupported engine option.

9.2.47 SL_ROLLOFFMODEL

```
#define SL_ROLLOFFMODEL_EXPONENTIAL    ((SLuint32) 0x00000000)
#define SL_ROLLOFFMODEL_LINEAR         ((SLuint32) 0x00000001)
```

These two macros define the two supported distance models: exponential and linear. The exponential distance model most closely models real-life, with an exponential decay due to distance from the listener. The linear distance model offers an alternative rolloff, in which the rate of attenuation is linearly proportional to the distance from the listener.

Value	Description
SL_ROLLOFFMODEL_EXPONENTIAL	Exponential distance rolloff model.
SL_ROLLOFFMODEL_LINEAR	Linear distance rolloff model.

9.2.48 SL_SAMPLINGRATE

```
#define SL_SAMPLINGRATE_8         ((SLuint32) 8000000)
#define SL_SAMPLINGRATE_11_025    ((SLuint32) 11025000)
#define SL_SAMPLINGRATE_12        ((SLuint32) 12000000)
#define SL_SAMPLINGRATE_16        ((SLuint32) 16000000)
#define SL_SAMPLINGRATE_22_05     ((SLuint32) 22050000)
#define SL_SAMPLINGRATE_24        ((SLuint32) 24000000)
#define SL_SAMPLINGRATE_32        ((SLuint32) 32000000)
#define SL_SAMPLINGRATE_44_1      ((SLuint32) 44100000)
#define SL_SAMPLINGRATE_48        ((SLuint32) 48000000)
#define SL_SAMPLINGRATE_64        ((SLuint32) 64000000)
#define SL_SAMPLINGRATE_88_2      ((SLuint32) 88200000)
#define SL_SAMPLINGRATE_96        ((SLuint32) 96000000)
#define SL_SAMPLINGRATE_192       ((SLuint32) 192000000)
```

These macros specify the commonly used sampling rates (in milliHertz) supported by most audio I/O devices.

Value	Description
SL_SAMPLINGRATE_8	8 kHz sampling rate.
SL_SAMPLINGRATE_11_025	11.025 kHz sampling rate.
SL_SAMPLINGRATE_12	12 kHz sampling rate.
SL_SAMPLINGRATE_16	16 kHz sampling rate.
SL_SAMPLINGRATE_22_05	22.05 kHz sampling rate.
SL_SAMPLINGRATE_24	24 kHz sampling rate.
SL_SAMPLINGRATE_32	32 kHz sampling rate.
SL_SAMPLINGRATE_44_1	44.1 kHz sampling rate.
SL_SAMPLINGRATE_48	48 kHz sampling rate.
SL_SAMPLINGRATE_64	64 kHz sampling rate.
SL_SAMPLINGRATE_88_2	88.2 kHz sampling rate.
SL_SAMPLINGRATE_96	96 kHz sampling rate.
SL_SAMPLINGRATE_192	192 kHz sampling rate.

9.2.49 SL_SEEKMODE

```
#define SL_SEEKMODE_FAST          ((SLuint32) 0x0001)
#define SL_SEEKMODE_ACCURATE      ((SLuint32) 0x0002)
```

These values represent seek modes.

The nature of encoded content and of the API implementation may imply tradeoffs between the accuracy and speed of a seek operation. Seek modes afford the application a means to specify which characteristic, accuracy or speed, should be preferred.

Value	Description
SL_SEEKMODE_FAST	Prefer the speed of a seek over the accuracy of a seek. Upon a `SetPosition()` call, the implementation minimizes latency potentially at the expense of accuracy; effective playback head position may vary slightly from the requested position
SL_SEEKMODE_ACCURATE	Prefer the accuracy of a seek over the speed of a seek. Upon a `SetPosition()` call, the implementation minimizes the distance between the effective playback head position and the requested position, potentially at the price of higher latency.

9.2.50 SL_SPEAKER

```
#define SL_SPEAKER_FRONT_LEFT                ((SLuint32) 0x00000001)
#define SL_SPEAKER_FRONT_RIGHT               ((SLuint32) 0x00000002)
#define SL_SPEAKER_FRONT_CENTER              ((SLuint32) 0x00000004)
#define SL_SPEAKER_LOW_FREQUENCY             ((SLuint32) 0x00000008)
#define SL_SPEAKER_BACK_LEFT                 ((SLuint32) 0x00000010)
#define SL_SPEAKER_BACK_RIGHT                ((SLuint32) 0x00000020)
#define SL_SPEAKER_FRONT_LEFT_OF_CENTER      ((SLuint32) 0x00000040)
#define SL_SPEAKER_FRONT_RIGHT_OF_CENTER     ((SLuint32) 0x00000080)
#define SL_SPEAKER_BACK_CENTER               ((SLuint32) 0x00000100)
#define SL_SPEAKER_SIDE_LEFT                 ((SLuint32) 0x00000200)
#define SL_SPEAKER_SIDE_RIGHT                ((SLuint32) 0x00000400)
#define SL_SPEAKER_TOP_CENTER                ((SLuint32) 0x00000800)
#define SL_SPEAKER_TOP_FRONT_LEFT            ((SLuint32) 0x00001000)
#define SL_SPEAKER_TOP_FRONT_CENTER          ((SLuint32) 0x00002000)
#define SL_SPEAKER_TOP_FRONT_RIGHT           ((SLuint32) 0x00004000)
#define SL_SPEAKER_TOP_BACK_LEFT             ((SLuint32) 0x00008000)
#define SL_SPEAKER_TOP_BACK_CENTER           ((SLuint32) 0x00010000)
#define SL_SPEAKER_TOP_BACK_RIGHT            ((SLuint32) 0x00020000)
```

Speaker location macros used when specifying a channel mask.

Value	Description
SL_SPEAKER_FRONT_LEFT	Front left speaker channel.
SL_SPEAKER_FRONT_RIGHT	Front right speaker channel.
SL_SPEAKER_FRONT_CENTER	Front center speaker channel.
SL_SPEAKER_LOW_FREQUENCY	Low frequency effects (LFE) speaker channel.

Value	Description
SL_SPEAKER_BACK_LEFT	Rear left speaker channel.
SL_SPEAKER_BACK_RIGHT	Rear right speaker channel.
SL_SPEAKER_FRONT_LEFT_OF_CENTER	Front left-of-center speaker channel.
SL_SPEAKER_FRONT_RIGHT_OF_CENTER	Front right-of-center speaker channel.
SL_SPEAKER_BACK_CENTER	Rear center speaker channel.
SL_SPEAKER_SIDE_LEFT	Side left speaker channel.
SL_SPEAKER_SIDE_RIGHT	Side right speaker channel.
SL_SPEAKER_TOP_CENTER	Top center speaker channel.
SL_SPEAKER_TOP_FRONT_LEFT	Top front left speaker channel.
SL_SPEAKER_TOP_FRONT_CENTER	Top front center speaker channel.
SL_SPEAKER_TOP_FRONT_RIGHT	Top front right speaker channel.
SL_SPEAKER_TOP_BACK_LEFT	Top rear left speaker channel.
SL_SPEAKER_TOP_BACK_CENTER	Top rear center speaker channel.
SL_SPEAKER_TOP_BACK_RIGHT	Top rear right speaker channel.

9.2.51 SL_TIME

```
#define SL_TIME_UNKNOWN    ((SLuint32) 0xFFFFFFFF)
```

These values are reserved for special designations of playback time that cannot be represented using the normal numeric range.

Value	Description
SL_TIME_UNKNOWN	The time is unknown (e.g duration of content in a broadcast stream)

9.2.52 SL_VOICETYPE

```
#define SL_VOICETYPE_2D_AUDIO          ((SLuint16) 0x0001)
#define SL_VOICETYPE_MIDI              ((SLuint16) 0x0002)
#define SL_VOICETYPE_3D_AUDIO          ((SLuint16) 0x0004)
#define SL_VOICETYPE_3D_MIDIOUTPUT     ((SLuint16) 0x0008)
```

These macros list the types of “voices” supported by the system (and not the number of voices of each type).

Value	Description
SL_VOICETYPE_2D_AUDIO	2D voices (normal non-3D sampled audio voices). Effectively refers to the mixer inputs supported by the system.
SL_VOICETYPE_MIDI	MIDI voices. Refers to MIDI polyphony.
SL_VOICETYPE_3D_AUDIO	3D voices (3D sampled audio).
SL_VOICETYPE_3D_MIDIOUTPUT	MIDI synthesizer output that can be 3D spatialized.

PART 3: APPENDICES

Appendix A: References

CP	OpenMAX Content Pipe Application Programming Interface Specification, The Khronos Group
DLS2	Downloadable Sounds Level 2.1 Specification (RP-025/Amd1), MIDI Manufacturers Association, Los Angeles, CA, USA, January 2001.
I3DL1	3D Audio Rendering and Evaluation Guidelines, 3D Working Group, Interactive Audio Special Interest Group, MIDI Manufacturers Association, Los Angeles, CA, June 9, 1998.
I3DL2	Interactive 3D Audio Rendering Guidelines, Level 2.0, 3D Working Group, Interactive Audio Special Interest Group, MIDI Manufacturers Association, Los Angeles, CA, September 20, 1999.
ISO31-11	Quantities and units – Part 11 : Mathematical signs and symbols for use in the physical sciences and technology, ISO 31-11 :1992.
ISO639	Language codes, http://www.iso.org/iso/en/prods-services/popstds/languagecodes.html, ISO 639.
ISO1000	SI units and recommendations for the use of their multiples and of certain other units, ISO 1000:1992, 2003.
ISO3166	Country name codes, http://www.iso.org/iso/en/prods-services/popstds/countrynamecodes.html, ISO 3166-1:2006
JSR135	JSR-135: Mobile Media API (http://www.jcp.org/en/jsr/detail?id=135).
mDLS	Mobile DLS Specification, RP-041, MIDI Manufacturers Association, Los Angeles, CA, USA, 2003.
MIDI	The Complete MIDI 1.0 Detailed Specification, Document version 96.1, MIDI Manufacturers Association, Los Angeles, CA, USA, 1996 (Contains MIDI 1.0 Detailed Specification, MIDI Time Code, Standard MIDI Files 1.0, General MIDI System Level 1, MIDI Show Control 1.1, and MIDI Machine Control)
MPEG1	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 11172-3 “Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, Part 3: Audio”, 1993
MPEG2	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 13818-3 “Information Technology - Generic Coding of Moving Pictures and Associated Audio, Part 3: Audio”, 1998.
mXMF	Mobile XMF Content Format Specification, RP-042. MIDI Manufacturers Association, Los Angeles, CA, USA, September 2004.
OMXAL	OpenMAX Application Layer Application Programming Interface Specification, The Khronos Group.

RFC3066	Tags for the Identifications of Languages, http://tools.ietf.org/html/rfc3066, RFC-3066, IETF, 2001.
SP-MIDI	Scalable Polyphony MIDI Specification (RP-034), MIDI Manufacturers Association, Los Angeles, CA, USA, December 2001.
UUID	Information Technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: Generation and Registration of Universally Unique Identifiers (UUIDs) and their Use as ASN.1 Object Identifier Components, ITU-T Rec. X.667 | ISO/IEC 9834-8, 2004

Appendix B: Sample Code

The sample code provided in this appendix shows how to use different interfaces. These code fragments are not necessarily complete. See Appendix C: for more complete examples.

B.1 Audio Playback and recording

B.1.1 Buffer Queue

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

#define SLEEP(x) /* Client system sleep function to sleep x milliseconds
would replace SLEEP macro */

#define MAX_NUMBER_INTERFACES 3
#define MAX_NUMBER_OUTPUT_DEVICES 6

/* Local storage for Audio data in 16 bit words */
#define AUDIO_DATA_STORAGE_SIZE 4096
/* Audio data buffer size in 16 bit words. 8 data segments are used in
this simple example */
#define AUDIO_DATA_BUFFER_SIZE 4096/8

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      // Debug printing to be placed here
      exit(1);
   }
}

/* Structure for passing information to callback function */
typedef struct CallbackCntxt_ {
   SLPlayItf  playItf;
   SLint16*   pDataBase;    // Base adress of local audio data storage
   SLint16*   pData;        // Current adress of local audio data storage
   SLuint32   size;
} CallbackCntxt;

/* Local storage for Audio data */
SLint16 pcmData[AUDIO_DATA_STORAGE_SIZE];

/* Callback for Buffer Queue events */
```

```
void BufferQueueCallback(
   SLBufferQueueItf queueItf,
   SLuint32 eventFlags,
   const void * pBuffer,
   SLuint32 bufferSize,
   SLuint32 dataUsed,
   void *pContext)
{
   SLresult res;
   CallbackCntxt *pCntxt = (CallbackCntxt*)pContext;
   if(pCntxt->pData < (pCntxt->pDataBase + pCntxt->size))
   {
      res = (*queueItf)->Enqueue(queueItf, (void*) pCntxt->pData,
             2 * AUDIO_DATA_BUFFER_SIZE, SL_BOOLEAN_FALSE); /* Size given
in bytes. */
      CheckErr(res);
      /* Increase data pointer by buffer size */
      pCntxt->pData += AUDIO_DATA_BUFFER_SIZE;
   }
}

/* Play some music from a buffer queue  */
void TestPlayMusicBufferQueue( SLObjectItf sl )
{
   SLEngineItf                 EngineItf;

   SLint32                     numOutputs = 0;
   SLuint32                    deviceID = 0;

   SLresult                    res;

   SLDataSource                audioSource;
   SLDataLocator_BufferQueue   bufferQueue;
   SLDataFormat_PCM            pcm;

   SLDataSink                  audioSink;
   SLDataLocator_OutputMix     locator_outputmix;

   SLObjectItf                 player;
   SLPlayItf                   playItf;
   SLBufferQueueItf            bufferQueueItf;
   SLBufferQueueState          state;

   SLObjectItf                 OutputMix;
   SLVolumeItf                 volumeItf;

   int                         i;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   /* Callback context for the buffer queue callback function */
   CallbackCntxt cntxt;
```

```
    /* Get the SL Engine Interface which is implicit */
    res = (*sl)->GetInterface(sl, SL_IID_ENGINE, (void*)&EngineItf);
    CheckErr(res);

    /* Initialize arrays required[] and iidArray[] */
    for (i=0;i<MAX_NUMBER_INTERFACES;i++)
    {
       required[i] = SL_BOOLEAN_FALSE;
       iidArray[i] = SL_IID_NULL;
    }

    // Set arrays required[] and iidArray[] for VOLUME interface
    required[0] = SL_BOOLEAN_TRUE;
    iidArray[0] = SL_IID_VOLUME;
    // Create Output Mix object to be used by player
    res = (*EngineItf)->CreateOutputMix(EngineItf, &OutputMix, 1,
          iidArray, required); CheckErr(res);

    // Realizing the Output Mix object in synchronous mode.
    res = (*OutputMix)->Realize(OutputMix, SL_BOOLEAN_FALSE);
    CheckErr(res);

    res = (*OutputMix)->GetInterface(OutputMix, SL_IID_VOLUME,
          (void*)&volumeItf); CheckErr(res);

    /* Setup the data source structure for the buffer queue */
    bufferQueue.locatorType = SL_DATALOCATOR_BUFFERQUEUE;
    bufferQueue.numBuffers = 4;  /* Four buffers in our buffer queue */

    /* Setup the format of the content in the buffer queue */
    pcm.formatType = SL_DATAFORMAT_PCM;
    pcm.numChannels = 2;
    pcm.sampleRate = SL_SAMPLINGRATE_44_1;
    pcm.bitsPerSample = SL_PCMSAMPLEFORMAT_FIXED_16;
    pcm.containerSize = 16;
    pcm.channelMask = SL_SPEAKER_FRONT_LEFT | SL_SPEAKER_FRONT_RIGHT;
    pcm.endianness = SL_BYTEORDER_LITTLEENDIAN;

    audioSource.pFormat      = (void *)&pcm;
    audioSource.pLocator     = (void *)&bufferQueue;

    /* Setup the data sink structure */
    locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
    locator_outputmix.outputMix    = OutputMix;
    audioSink.pLocator           = (void *)&locator_outputmix;
    audioSink.pFormat            = NULL;

    /* Initialize the context for Buffer queue callbacks */
    cntxt.pDataBase = (void*)&pcmData;
    cntxt.pData = cntxt.pDataBase;
    cntxt.size = sizeof(pcmData);
```

```
   /* Set arrays required[] and iidArray[] for SEEK interface
     (PlayItf is implicit) */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_BUFFERQUEUE;

   /* Create the music player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, &player,
       &audioSource, &audioSink, 1, iidArray, required); CheckErr(res);

   /* Realizing the player in synchronous mode. */
   res = (*player)->Realize(player, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Get seek and play interfaces */
   res = (*player)->GetInterface(player, SL_IID_PLAY, (void*)&playItf);
   CheckErr(res);

   res = (*player)->GetInterface(player, SL_IID_BUFFERQUEUE,
         (void*)&bufferQueueItf); CheckErr(res);

   /* Setup to receive buffer queue event callbacks */
   res = (*bufferQueueItf)->RegisterCallback(bufferQueueItf,
         BufferQueueCallback, NULL); CheckErr(res);

   /* Before we start set volume to -3dB (-300mB) */
   res = (*volumeItf)->SetVolumeLevel(volumeItf, -300); CheckErr(res);

   /* Enqueue a few buffers to get the ball rolling */
   res = (*bufferQueueItf)->Enqueue(bufferQueueItf, cntxt.pData,
          2 * AUDIO_DATA_BUFFER_SIZE, SL_BOOLEAN_FALSE); /* Size given in
bytes. */
   CheckErr(res);
   cntxt.pData += AUDIO_DATA_BUFFER_SIZE;

   res = (*bufferQueueItf)->Enqueue(bufferQueueItf, cntxt.pData,
          2 * AUDIO_DATA_BUFFER_SIZE, SL_BOOLEAN_FALSE); /* Size given in
bytes. */
   CheckErr(res);
   cntxt.pData += AUDIO_DATA_BUFFER_SIZE;

   res = (*bufferQueueItf)->Enqueue(bufferQueueItf, cntxt.pData,
          2 * AUDIO_DATA_BUFFER_SIZE, SL_BOOLEAN_FALSE); /* Size given in
bytes. */
   CheckErr(res);
   cntxt.pData += AUDIO_DATA_BUFFER_SIZE;

   /* Play the PCM samples using a buffer queue */
   res = (*playItf)->SetPlayState( playItf, SL_PLAYSTATE_PLAYING );
   CheckErr(res);

   /* Wait until the PCM data is done playing, the buffer queue callback
      will continue to queue buffers until the entire PCM data has been
      played. This is indicated by waiting for the count member of the
      SLBufferQueueState to go to zero.
   */
   res = (*bufferQueueItf)->GetState(bufferQueueItf, &state);
   CheckErr(res);

   while(state.count)
   {
      (*bufferQueueItf)->GetState(bufferQueueItf, &state);
   }

   /* Make sure player is stopped */
   res = (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_STOPPED);
   CheckErr(res);
   /* Destroy the player */
   (*player)->Destroy(player);
   /* Destroy Output Mix object */
   (*OutputMix)->Destroy(OutputMix);
}

int sl_main( void )
{
    SLresult    res;
    SLObjectItf sl;

    SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };


    res = slCreateEngine( &sl, 3, EngineOption, 0, NULL, NULL);
    CheckErr(res);
    /* Realizing the SL Engine in synchronous mode. */
    res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
    TestPlayMusicBufferQueue(sl);
    /* Shutdown OpenSL ES */
    (*sl)->Destroy(sl);
    exit(0);
}
```

B.1.2 Recording

Audio recorder example.

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

#define MAX_NUMBER_INTERFACES 5
#define MAX_NUMBER_INPUT_DEVICES 3
```

```
#define POSITION_UPDATE_PERIOD 1000 /* 1 sec */

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      // Debug printing to be placed here
      exit(1);
   }
}

void RecordEventCallback(SLRecordItf caller,
                                  void      *pContext,
                                  SLuint32  recordevent)
{
   /* Callback code goes here */
}


/*
 * Test recording of audio from a microphone into a specified file
 */
void TestAudioRecording(SLObjectItf sl)
{
   SLObjectItf         recorder;
   SLRecordItf         recordItf;
   SLEngineItf         EngineItf;
   SLAudioIODeviceCapabilitiesItf AudioIODeviceCapabilitiesItf;
   SLAudioInputDescriptor        AudioInputDescriptor;
   SLresult                res;

   SLDataSource            audioSource;
   SLDataLocator_IODevice  locator_mic;
   SLDeviceVolumeItf       devicevolumeItf;

   SLDataSink              audioSink;
   SLDataLocator_URI       uri;
   SLDataFormat_MIME       mime;

   int                     i;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   SLuint32 InputDeviceIDs[MAX_NUMBER_INPUT_DEVICES];
   SLint32   numInputs = 0;
   SLboolean mic_available = SL_BOOLEAN_FALSE;
   SLuint32 mic_deviceID = 0;

   /* Get the SL Engine Interface which is implicit */
   res = (*sl)->GetInterface(sl, SL_IID_ENGINE, (void*)&EngineItf);
CheckErr(res);
```

```
   /* Get the Audio IO DEVICE CAPABILITIES interface, which is also
implicit */
   res = (*sl)->GetInterface(sl, SL_IID_AUDIOIODEVICECAPABILITIES,
(void*)&AudioIODeviceCapabilitiesItf); CheckErr(res);
   numInputs = MAX_NUMBER_INPUT_DEVICES;

   res = (*AudioIODeviceCapabilitiesItf)->GetAvailableAudioInputs(
AudioIODeviceCapabilitiesItf, &numInputs, InputDeviceIDs); CheckErr(res);

   /* Search for either earpiece microphone or headset microphone input
device - with a preference for the latter */
   for (i=0;i<numInputs; i++)
   {
      res = (*AudioIODeviceCapabilitiesItf)-
>QueryAudioInputCapabilities(AudioIODeviceCapabilitiesItf,
InputDeviceIDs[i], &AudioInputDescriptor); CheckErr(res);
      if((AudioInputDescriptor.deviceConnection ==
SL_DEVCONNECTION_ATTACHED_WIRED)&&
          (AudioInputDescriptor.deviceScope == SL_DEVSCOPE_USER)&&
          (AudioInputDescriptor.deviceLocation ==
SL_DEVLOCATION_HEADSET))
      {
         mic_deviceID = InputDeviceIDs[i];
         mic_available = SL_BOOLEAN_TRUE;
      break;
      }
      else if((AudioInputDescriptor.deviceConnection ==
SL_DEVCONNECTION_INTEGRATED)&&
                  (AudioInputDescriptor.deviceScope ==
SL_DEVSCOPE_USER)&&
                  (AudioInputDescriptor.deviceLocation ==
SL_DEVLOCATION_HANDSET))
      {
         mic_deviceID = InputDeviceIDs[i];
         mic_available = SL_BOOLEAN_TRUE;
      break;
      }
   }

   /* If neither of the preferred input audio devices is available, no
point in continuing */
   if (!mic_available) {
      /* Appropriate error message here */
      exit(1);
   }

   /* Initialize arrays required[] and iidArray[] */
   for (i=0;i<MAX_NUMBER_INTERFACES;i++)
   {
      required[i] = SL_BOOLEAN_FALSE;
      iidArray[i] = SL_IID_NULL;
   }
   /* Get the optional DEVICE VOLUME interface from the engine */
   res = (*sl)->GetInterface(sl, SL_IID_DEVICEVOLUME,
(void*)&devicevolumeItf); CheckErr(res);

   /* Set recording volume of the microphone to -3 dB */
   res = (*devicevolumeItf)->SetVolume(devicevolumeItf, mic_deviceID, -
300); CheckErr(res);

   /* Setup the data source structure */
   locator_mic.locatorType   = SL_DATALOCATOR_IODEVICE;
   locator_mic.deviceType    = SL_IODEVICE_AUDIOINPUT;
   locator_mic.deviceID          = mic_deviceID;
   locator_mic.device      = NULL;
   audioSource.pLocator      = (void *)&locator_mic;
   audioSource.pFormat       = NULL;

   /* Setup the data sink structure */
   uri.locatorType         = SL_DATALOCATOR_URI;
   uri.pURI                = (SLchar *) "file:///recordsample.wav";
   mime.formatType         = SL_DATAFORMAT_MIME;
   mime.pMimeType          = (SLchar *) "audio/x-wav";
   mime.containerType      = SL_CONTAINERTYPE_WAV;
   audioSink.pLocator      = (void *)&uri;
   audioSink.pFormat       = (void *)&mime;

   /* Create audio recorder */
   res = (*EngineItf)->CreateAudioRecorder(EngineItf, &recorder,
&audioSource, &audioSink, 0, iidArray, required); CheckErr(res);

   /* Realizing the recorder in synchronous mode. */
   res = (*recorder)->Realize(recorder, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Get the RECORD interface - it is an implicit interface */
   res = (*recorder)->GetInterface(recorder, SL_IID_RECORD,
(void*)&recordItf); CheckErr(res);

   /* Setup to receive position event callbacks */
   res = (*recordItf)->RegisterCallback(recordItf, RecordEventCallback,
NULL);
   CheckErr(res);

   /* Set notifications to occur after every second - may be useful in
updating a recording progress bar */
   res = (*recordItf)->SetPositionUpdatePeriod( recordItf,
POSITION_UPDATE_PERIOD); CheckErr(res);
   res = (*recordItf)->SetCallbackEventsMask( recordItf,
SL_RECORDEVENT_HEADATNEWPOS); CheckErr(res);


   /* Set the duration of the recording - 30 seconds (30,000
milliseconds) */
   res = (*recordItf)->SetDurationLimit(recordItf, 30000); CheckErr(res);
   /* Record the audio */
   res = (*recordItf)-
>SetRecordState(recordItf,SL_RECORDSTATE_RECORDING);


   /* Destroy the recorder object */
   (*recorder)->Destroy(recorder);
}

int sl_main( void )
{
    SLresult    res;
    SLObjectItf sl;

    /* Create OpenSL ES engine in thread-safe mode */
    SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };


    res = slCreateEngine( &sl, 3, EngineOption, 0, NULL, NULL);
CheckErr(res);

    /* Realizing the SL Engine in synchronous mode. */
    res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
    TestAudioRecording(sl);
    /* Shutdown OpenSL ES */
    (*sl)->Destroy(sl);
    exit(0);
}
```

B.2 Dynamic Interface Management

```
/* Example callback for dynamic interface management events. *
 * See section 3.3 in the specification for information on   *
 * what operations are allowed in callbacks.                 *
 *                                                           *
 * The following example consists of pseudo code and is not  *
 * intended to be compiled. It is provided as a starting     *
 * point for developing compilable code.                     */
void DIMCallback(    SLDynamicInterfaceManagementItf  caller,
                     void                            *pContext,
                     SLuint32                         event,
                     SLresult                         result,
                     const SLInterfaceId              iid)
{
  if(event==SL_DYNAMIC_ITF_EVENT_ASYNC_TERMINATION) {
      if((iid == SL_IID_PRESETREVERB)){
        asyncRes=result;
        sem_post(&semDIM);
      }
  } else {
    event_post(eventQueue, {caller, pContext, event, result, iid});
  }
}

/* Example main event loop thread for handling DIM events */
void eventThread()
{
  SLresult res;
  while(...) /* Event loop */
  {
    event=event_read(eventQueue);
    switch(event.event) {
    case SL_DYNAMIC_ITF_EVENT_RESOURCES_LOST:
      if((event.iid == SL_IID_PRESETREVERB)){
         presetReverbHasResources=SL_BOOLEAN_FALSE;
      }
      break;
    case SL_DYNAMIC_ITF_EVENT_RESOURCES_LOST_PERMANENTLY:
      if((event.iid == SL_IID_PRESETREVERB)) {
         presetReverbHasResources=SL_BOOLEAN_FALSE;
         /* Dynamically remove the PresetReverb interface from the
          * Output Mix object since this instance will not be
          * useful anymore. */
         res = (*(event.caller))->RemoveInterface(event.caller,
               SL_IID_PRESETREVERB); CheckErr(res);
         presetReverbIsRealized = SL_BOOLEAN_FALSE;
      }
      break;
    case SL_DYNAMIC_ITF_EVENT_RESOURCES_AVAILABLE:
      if((event.iid == SL_IID_PRESETREVERB) && (presetReverbIsRealized))
{
         /* Dynamically resume the PresetReverb interface
            on the Output Mix object */
         res = (*(event.caller))->ResumeInterface(event.caller,
               SL_IID_PRESETREVERB); CheckErr(res);
         /* Wait until asynchronous call terminates */
         sem_wait(&semDIM);
         if (asyncRes == SL_RESULT_SUCCESS) {
           /* We got the resource */
           presetReverbHasResources=SL_BOOLEAN_TRUE;
         } else {
           /* Some other interface beat us to claiming the available
            resource, lets wait for a new event */
           presetReverbHasResources=SL_BOOLEAN_FALSE;
         }
      }
      break;
    default:
      break;
    }
  }
}

/* Example program using dynamic interface management interface. */
int sl_main()
{
  SLDynamicInterfaceManagementItf dynamicInterfaceManagementItf;
  /* ...
    Start event thread
    Create Output Mix object to be used by player */
  res = (*EngineItf)->CreateOutputMix(EngineItf, &OutputMix, 1,
         iidArray, required); CheckErr(res);
  /* Realizing the Output Mix object in synchronous mode. */
  res = (*OutputMix)->Realize(OutputMix, SL_BOOLEAN_FALSE);
  CheckErr(res);
  /* Get the Dynamic Interface Management interface for the
     Output Mix object */
  res = (*OutputMix)->GetInterface(OutputMix,
         SL_IID_DYNAMICINTERFACEMANAGEMENT,
         (void*)&dynamicInterfaceManagementItf); CheckErr(res);
  /* Register DIM callback */
  res = (*dynamicInterfaceManagementItf)->
         RegisterCallback(dynamicInterfaceManagementItf, DIMCallback,
                          NULL); CheckErr(res);
  while(...) /* main action loop */
  {
    switch(action) {
    case 'AddReverb':
      /* Add reverb if not already present and
         there are sufficient resources. */
      if (!presetReverbIsRealized) {
          /* We should stop playback first to increase
             chances of success.
             Dynamically add the PresetReverb interface
             to the Output Mix object */
          res = (*dynamicInterfaceManagementItf)->
                AddInterface(dynamicInterfaceManagementItf,
                     SL_IID_PRESETREVERB); CheckErr(res);
          /* Wait until asynchronous call terminates */
          sem_wait(&semDIM);
          if (asyncRes == SL_RESULT_SUCCESS) {
              presetReverbHasResources = SL_BOOLEAN_TRUE;
              presetReverbIsRealized = SL_BOOLEAN_TRUE;
              /* Get PresetReverb interface */
              res = (*OutputMix)->GetInterface(OutputMix,
                     SL_IID_PRESETREVERB, (void*)&PresetReverbItf);
              CheckErr(res);
              /* Setup PresetReverb for LARGE HALL */
              res = (*PresetReverbItf)->SetPreset(PresetReverbItf,
                     SL_REVERBPRESET_LARGEHALL); CheckErr(res);
          } else if(asyncRes == SL_RESULT_RESOURCE_ERROR){
              /* Did not get resources now,
                 will get callback when resources are available */
              presetReverbHasResources = SL_BOOLEAN_FALSE;
              presetReverbIsRealized = SL_BOOLEAN_TRUE;
          } else {
              /* Did NOT successfully add presetReverb
                 to the output mix object */
              presetReverbHasResources = SL_BOOLEAN_FALSE;
              presetReverbIsRealized = SL_BOOLEAN_FALSE;
          }
          /* Start playback again. */
      }
      break;
    case 'RemoveReverb':
      /* Remove the Preset Reverb if present. */
      if (presetReverbIsRealized) {
          /* Dynamically remove the PresetReverb interface
             from the Output Mix object */
          presetReverbIsRealized = SL_BOOLEAN_FALSE;
          presetReverbHasResources = SL_BOOLEAN_FALSE;
          res = (*dynamicInterfaceManagementItf)->
                RemoveInterface(dynamicInterfaceManagementItf,
                SL_IID_PRESETREVERB); CheckErr(res);
      }
      break;
    }
  }
}
```

B.3 MIDI

B.3.1 Simple MIDI

Tests the basic load of a MIDI file from a standard locator.

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      // Debug printing to be placed here
      exit(1);
   }
}

void TestMIDISimple( SLEngineItf eng, SLObjectItf outputMix )
{
    SLresult          res;
    SLDataSource      fileSrc;
    SLDataSource      bankSrc;
    SLDataSink        audOutSnk;
    SLObjectItf       player;
    SLmillisecond     dur;
    SLmillisecond     pos;
    SLPlayItf         playItf;

    SLDataLocator_URI       fileLoc = { SL_DATALOCATOR_URI, (SLchar *)
"file:///foo.mid" };
    SLDataFormat_MIME       fileFmt = { SL_DATAFORMAT_MIME, (SLchar *)
"audio/x-midi", SL_CONTAINERTYPE_SMF };
    SLDataLocator_URI       bankLoc = { SL_DATALOCATOR_URI, (SLchar *)
"file:///foo.dls" };
    SLDataFormat_MIME       bankFmt = { SL_DATAFORMAT_MIME, (SLchar *)
"audio/dls", SL_CONTAINERTYPE_MOBILE_DLS };
    SLDataLocator_OutputMix  audOutLoc;

   const SLboolean       required[2] = {SL_BOOLEAN_TRUE,
SL_BOOLEAN_FALSE};
    const SLInterfaceID   iidArray[2] = {SL_IID_PLAY, SL_IID_VOLUME};

    res = (*eng)->CreateOutputMix(eng, &outputMix, 1, &iidArray[1],
&required[1]); CheckErr(res);

    res = (*outputMix)->Realize(outputMix, SL_BOOLEAN_FALSE);

    fileSrc.pFormat = &fileFmt;
    fileSrc.pLocator = &fileLoc;
    bankSrc.pFormat = &bankFmt;
    bankSrc.pLocator = &bankLoc;
    audOutLoc.locatorType = SL_DATALOCATOR_OUTPUTMIX;
    audOutLoc.outputMix = outputMix;
    audOutSnk.pFormat = NULL;
    audOutSnk.pLocator = &audOutLoc;
    res = (*eng)->CreateMidiPlayer(eng, &player, &fileSrc, &bankSrc,
&audOutSnk, NULL, NULL, 1, iidArray, required); CheckErr(res);
    res = (*player)->Realize(player, SL_BOOLEAN_FALSE); CheckErr(res);

    /* Get the play interface, which was implicitly created on the
    MIDI player creation. */
    res = (*player)->GetInterface(player, SL_IID_PLAY, (void *)&playItf);
CheckErr(res);

    res = (*playItf)->GetDuration(playItf, &dur); CheckErr(res);
    res = (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);
    do
    {
        res = (*playItf)->GetPosition(playItf, &pos); CheckErr(res);
    } while(pos < dur);

    /* Destroy player */
    (*player)->Destroy(player);
}
```

B.3.2 MIDI Buffer Queue

Tests the MIDI buffer queue. This is in an OpenSL ES game context.

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

void *queueData[8] = { (void *) 0x123, (void *) 0x234, (void *) 0x345,
(void *) 0x456, (void *) 0x567, (void *) 0x678, (void *) 0x789 };
SLuint32 index = 0;

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      // Debug printing to be placed here
      exit(1);
   }
}
```

```
void TestQueueCallback(SLBufferQueueItf caller, SLuint32 eventFlags,
const void* pBuffer, SLuint32 bufferSize, SLuint32 dataUsed, void
*pContext)
{
    SLBufferQueueItf queueItf = (SLBufferQueueItf) pContext;
    SLresult res = (*queueItf)->Enqueue(queueItf, queueData[index++],
1024, SL_BOOLEAN_FALSE); CheckErr(res);
    index &= 0x7; /* force the queues to cycle */
}
void TestMIDIBufferQueue(SLEngineItf eng, SLObjectItf outputMix)
{
    SLresult        res;
    SLDataSource    midSrc;
    SLDataSource    bnkSrc;
    SLDataSink      audOutSnk;
    SLObjectItf     player;
    SLPlayItf       playItf;
    SLBufferQueueItf  queueItf;

    SLDataLocator_MIDIBufferQueue  midLoc;
    SLDataFormat_MIME              midFmt;
    SLDataLocator_URI              bankLoc;
    SLDataFormat_MIME              bankFmt;
    SLDataLocator_OutputMix        audOutLoc;

    const SLboolean       required[3] = { SL_BOOLEAN_TRUE,
SL_BOOLEAN_TRUE ,SL_BOOLEAN_FALSE};
    const SLInterfaceID   iidArray[3] = { SL_IID_PLAY,
SL_IID_BUFFERQUEUE ,SL_IID_VOLUME};

    res = (*eng)->CreateOutputMix(eng, &outputMix, 1, &iidArray[2],
&required[2]); CheckErr(res);
    res = (*outputMix)->Realize(outputMix, SL_BOOLEAN_FALSE);
CheckErr(res);

    /* Set up the MIDI buffer queue data source */
    midLoc.locatorType = SL_DATALOCATOR_MIDIBUFFERQUEUE;
    midLoc.tpqn = 96;
    midLoc.numBuffers = 3;
    midSrc.pLocator = &midLoc;
    midFmt.formatType = SL_DATAFORMAT_MIME;
    midFmt.pMimeType = (SLchar *) "audio/sp-midi";
    midFmt.containerType = SL_CONTAINERTYPE_SMF;
    midSrc.pFormat = &midFmt;

    /* Set up the bank data source */
    bankLoc.locatorType = SL_DATALOCATOR_URI;
    bankLoc.pURI = (SLchar *) "file:///foo.dls";
    bankFmt.formatType = SL_DATAFORMAT_MIME;
    bankFmt.pMimeType = (SLchar *) "audio/dls";
    bankFmt.containerType = SL_CONTAINERTYPE_MOBILE_DLS;
    bnkSrc.pFormat = &bankFmt;
    bnkSrc.pLocator = &bankLoc;
    /* Set up the audio output data sink */
    audOutLoc.locatorType = SL_DATALOCATOR_OUTPUTMIX;
    audOutLoc.outputMix = outputMix;
    audOutSnk.pFormat = NULL;
    audOutSnk.pLocator = &audOutLoc;

    /* Prepare and play the player */
    res = (*eng)->CreateMidiPlayer(eng, &player, &midSrc, &bnkSrc,
&audOutSnk, NULL, NULL, 2, iidArray, required); CheckErr(res);
    res = (*player)->Realize(player, SL_BOOLEAN_FALSE); CheckErr(res);
    res = (*player)->GetInterface(player, SL_IID_PLAY, (void *)&playItf);
CheckErr(res);
    res = (*player)->GetInterface(player, SL_IID_BUFFERQUEUE, (void
*)&queueItf); CheckErr(res);

    res = (*queueItf)->RegisterCallback(queueItf, TestQueueCallback,
(void *)&queueItf); CheckErr(res);

    /* Enqueue three buffers */
    res = (*queueItf)->Enqueue(queueItf, queueData[index++], 1024,
SL_BOOLEAN_FALSE); CheckErr(res);
    res = (*queueItf)->Enqueue(queueItf, queueData[index++], 1024,
SL_BOOLEAN_FALSE); CheckErr(res);
    res = (*queueItf)->Enqueue(queueItf, queueData[index++], 1024,
SL_BOOLEAN_FALSE); CheckErr(res);


    res = (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);
    {
        SLuint32 state = SL_PLAYSTATE_PLAYING;
        while(state == SL_PLAYSTATE_PLAYING)
        {
            res = (*playItf)->GetPlayState(playItf, &state);
CheckErr(res);
        }
    }
    /* Destroy player */
    (*player)->Destroy(player);
}
```

B.3.3 Advanced MIDI: MIDI messaging

Tests advanced features of the MIDI interfaces. This is in an OpenSL ES game context.

```
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

/* Checks for error. If any errors exit the application! */
```

```
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      // Debug printing to be placed here
      exit(1);
   }
}

void TestMetaEventCallback(SLMIDIMessageItf caller, void *pContext,
SLuint8 type, SLuint32 len, const SLuint8 *data, SLuint32 tick, SLuint16
track)
{
    if (pContext == (void *) 0x234 && *data++ == 0x02) /* check if it's a
copyright metadata item */
    {
        char str[256];
        SLuint8 stringLen = *data++;
        assert(stringLen == len - 2);
        memcpy(str, data, stringLen);
        printf("Copyright: %s from track %d at tick %d", str, track,
tick);
    }
}
void TestMIDIMessageCallback(SLMIDIMessageItf caller, void *pContext,
SLuint8 statusByte, SLuint32 length, const SLuint8 *data, SLuint32 tick,
SLuint16 track)
{
    if (pContext == (void *) 0x567 && statusByte >> 4 == 0xB)
    {
        if (statusByte >> 4 == 0xB)
        {
            printf("MIDI control change encountered at tick %d.
Channel %d; controller %d, value %d\n", tick, statusByte & 0xF0, data[0],
data[1]);
        }
        else if (statusByte >> 4 == 0xC)
        {
            printf("Program change encountered at tick %d. Channel %d;
program %d\n", tick, statusByte & 0xF0, data[0]);
        }
        else
        {
            printf("Error: unspecified MIDI event encountered in
TestMIDIMessageCallback\n");
        }
    }
}
void TestMIDIAdvanced(SLEngineItf eng, SLObjectItf outputMix)
{
    SLresult              res;
    SLDataSource          file;
    SLDataSource          bank;
    SLDataSink            audOutSnk;
    SLObjectItf           player;
    SLuint32              dur;                    /* duration in ticks */
    SLuint16              tracks;
    const SLboolean       required[7] = { SL_BOOLEAN_TRUE,
SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE,
SL_BOOLEAN_TRUE,SL_BOOLEAN_FALSE};
    const SLInterfaceID  iidArray[7] = { SL_IID_PLAY, SL_IID_SEEK,
SL_IID_MIDIMESSAGE, SL_IID_MIDIMUTESOLO, SL_IID_MIDITEMPO,
SL_IID_MIDITIME , SL_IID_VOLUME};
    SLPlayItf            playItf;
    SLSeekItf            seekItf;
    SLMIDIMessageItf   midMsgItf;
    SLMIDIMuteSoloItf  midMuteSoloItf;
    SLMIDITempoItf     midTempoItf;
    SLMIDITimeItf      midTimeItf;
    SLDataLocator_OutputMix  audOutLoc;

    SLDataLocator_URI  fileLoc = { SL_DATALOCATOR_URI, (SLchar *)
"file:///foo.mid" };
    SLDataFormat_MIME  fileFmt = { SL_DATAFORMAT_MIME, (SLchar *)
"audio/x-midi", SL_CONTAINERTYPE_SMF };
    SLDataLocator_URI  bankLoc = { SL_DATALOCATOR_URI, (SLchar *)
"file:///foo.dls" };
    SLDataFormat_MIME  bankFmt = { SL_DATAFORMAT_MIME, (SLchar *)
"audio/dls", SL_CONTAINERTYPE_MOBILE_DLS };
    res = (*eng)->CreateOutputMix(eng, &outputMix, 1, &iidArray[6],
&required[6]); CheckErr(res);
   res = (*outputMix)->Realize(outputMix, SL_BOOLEAN_FALSE);
CheckErr(res);

    file.pFormat = &fileFmt;
    file.pLocator = &fileLoc;
    bank.pFormat = &bankFmt;
    bank.pLocator = &bankLoc;
    audOutLoc.locatorType = SL_DATALOCATOR_OUTPUTMIX;
    audOutLoc.outputMix = outputMix;
    audOutSnk.pFormat = NULL;
    audOutSnk.pLocator = &audOutLoc;

    res = (*eng)->CreateMidiPlayer(eng, &player, &file, &bank,
&audOutSnk, NULL, NULL, 6, iidArray, required); CheckErr(res);
    res = (*player)->Realize(player, SL_BOOLEAN_FALSE); CheckErr(res);
    res = (*player)->GetInterface(player, iidArray[0], (void *)&playItf);
CheckErr(res);
    res = (*player)->GetInterface(player, iidArray[1], (void *)&seekItf);
CheckErr(res);
    res = (*player)->GetInterface(player, iidArray[2], (void
*)&midMsgItf); CheckErr(res);
    res = (*player)->GetInterface(player, iidArray[3], (void
*)&midMuteSoloItf); CheckErr(res);
    res = (*player)->GetInterface(player, iidArray[4], (void
*)&midTempoItf); CheckErr(res);
```

```
    res = (*player)->GetInterface(player, iidArray[5], (void
*)&midTimeItf); CheckErr(res);

    /* Set tempo to 140 BPM */
    res = (*midTempoItf)->SetMicrosecondsPerQuarterNote(midTempoItf,
428571); CheckErr(res);
    {
        /* Set channel volume on channel 3 to 96 */
        SLuint8 msg[3] = { 0xB0 | 0x02, 0x07, 0x60 };
        res = (*midMsgItf)->SendMessage(midMsgItf, msg, 3);
CheckErr(res);
    }
    {
        /* Set pitch bend sensitivity on channel 1 to +/- 1 semitone */
        SLuint8 msg[12] = { 0xB0 | 0x00, 0x65, 0x00,
                          /* RPN ID MSB (controller 101, data 0)  */
                         0xB0 | 0x00, 0x64, 0x00,
                         /* RPN ID LSB (controller 100, data 0)  */
                         0xB0 | 0x00, 0x06, 0x01,
                         /* RPN data MSB (controller 6, data 1)  */
                         0xB0 | 0x00, 0x26, 0x00 };
                         /* RPN data LSB (controller 38, data 0) */
        res = (*midMsgItf)->SendMessage(midMsgItf, msg, 12);
CheckErr(res);
    }
    /* Set/enable looping for arbitrary tick values (assume end <
duration) */
    res = (*seekItf)->SetLoop(seekItf, SL_BOOLEAN_TRUE, 0, 100);
CheckErr(res);

    /* Override loop points using MIDI tick precision */
    res = (*midTimeItf)->SetLoopPoints(midTimeItf, 1000, 6000);
CheckErr(res);
    /* Set a meta-event callback for the function TestMetaEventCallback
*/
    res = (*midMsgItf)->RegisterMetaEventCallback(midMsgItf,
TestMetaEventCallback, (void*) 0x234); CheckErr(res);

    /* Set a MIDI event callback for the function TestMIDIMessageCallback
*/
    res = (*midMsgItf)->RegisterMIDIMessageCallback(midMsgItf,
TestMIDIMessageCallback, (void *) 0x456); CheckErr(res);
    res = (*midMsgItf)->AddMIDIMessageCallbackFilter(midMsgItf,
SL_MIDIMESSAGETYPE_CONTROL_CHANGE); CheckErr(res);
    res = (*midMsgItf)->AddMIDIMessageCallbackFilter(midMsgItf,
SL_MIDIMESSAGETYPE_PROGRAM_CHANGE); CheckErr(res);

    /* Mute track 3 */
    res = (*midMuteSoloItf)->GetTrackCount(midMuteSoloItf, &tracks);
CheckErr(res);
    if (tracks > 2)
        res = (*midMuteSoloItf)->SetTrackMute(midMuteSoloItf, 2,
SL_BOOLEAN_TRUE); CheckErr(res);
```

```
    /* Get duration of the MIDI file in milliseconds */
    res = (*midTimeItf)->GetDuration(midTimeItf, &dur); CheckErr(res);

    /* Play half the MIDI data (tick duration / 2) */
    res = (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);
    {
        SLuint32 state;
        SLuint32 tickPos;
        do
        {
            SLEEP(100); /* sleep 100 ms */
            res = (*playItf)->GetPlayState(playItf, &state);
CheckErr(res);
            res = (*midTimeItf)->GetPosition(midTimeItf, &tickPos);
CheckErr(res);
        } while(tickPos < dur / 2 && state == SL_PLAYSTATE_PLAYING);
    }

    /* Destroy interfaces and player */
    (*player)->Destroy(player);
}
```

B.4 Metadata Extraction

B.4.1 Simple Metadata Extraction

Tests the basic features of metadata extraction.

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
       /* Debug printing to be placed here */
      exit(1);
   }
}

/*
 * Prints all ASCII metadata key-value pairs (from the root
 * of the media since MetadataTraversalItf is not used)
 */
void TestMetadataSimple(SLMetadataExtractionItf mdExtrItf)
{
   SLresult          res;
   SLuint32            mdCount   = 0;
   SLuint32            i;

   /* scan through the metadata items */
   res = (*mdExtrItf)->GetItemCount(mdExtrItf, &mdCount); CheckErr(res);
   for (i = 0; i < mdCount; ++i)
   {
      SLMetadataInfo *key = NULL;
      SLMetadataInfo *value = NULL;
      SLuint32 itemSize = 0;

      /* get the size of and malloc memory for the metadata item */
      res = (*mdExtrItf)->GetKeySize(mdExtrItf, i, &itemSize);
CheckErr(res);
      key = malloc(itemSize);
      if (key) /* no malloc error */
      {
         /* extract the key into the memory */
         res = (*mdExtrItf)->GetKey(mdExtrItf, i, itemSize, key);
CheckErr(res);
         if (key->encoding == SL_CHARACTERENCODING_ASCII)
         {
            res = (*mdExtrItf)->GetValueSize(mdExtrItf, i, &itemSize);
CheckErr(res);
            value = malloc(itemSize);
            if (value)  /* no malloc error */
            {
               /* extract the value into the memory */
               res = (*mdExtrItf)->GetValue(mdExtrItf, i, itemSize,
value); CheckErr(res);
               if (value->encoding == SL_CHARACTERENCODING_ASCII)
               {
                  printf("Item %d key: %s, value %s", i, key->data,
value->data);
               }
               free(value);
            }
         }
         free(key);
      }
   }
}
```

B.5 3D Audio

B.5.1 Simple 3D

An example showing how to create 3D source and spin it around the listener.

```
#include <stdio.h>
#include <stdlib.h>
/* Floating point routines sinf and cosf used in example - could be
replaced by fixed point code. */
#include <math.h>

#include <SLES/OpenSLES.h>

#define SLEEP(x)        // Client system sleep function to sleep x
// milliseconds would replace SLEEP macro

/*******************************************************************/

#define MAX_NUMBER_INTERFACES 2

#define CIRCLE_RADIUS      1000  /* 1.0 meters */
/* we move the source by this angle (in radians) at each step */
#define CIRCLE_STEP        (float) ( TWOPI / 180.0 / 2.0 )

#define PI                 3.1415926535f
#define TWOPI        ( 2.0f * PI )


/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      // Debug printing to be placed here
      exit(1);
   }
}

/* Play create a 3D source and spin it around the listener  */
void TestSimple3D( SLObjectItf sl )
{
   SLEngineItf             EngineItf;

   SLresult                res;

   SLDataSource            audioSource;
   SLDataLocator_URI       uri;
   SLDataFormat_MIME       mime;
   SLDataSink              audioSink;
   SLDataLocator_OutputMix locator_outputmix;

   SLObjectItf             player;
   SLPlayItf               playItf;
   SL3DLocationItf         locationItf;

   SLObjectItf             listener;

   SLObjectItf             OutputMix;

   int i;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   /* Get the SL Engine Interface which is implicit */
   res = (*sl)->GetInterface(sl, SL_IID_ENGINE, (void*)&EngineItf);
CheckErr(res);

   /* Initialize arrays required[] and iidArray[] */
   for (i=0;i<MAX_NUMBER_INTERFACES;i++)
   {
      required[i] = SL_BOOLEAN_FALSE;
      iidArray[i] = SL_IID_NULL;
   }

   /* Create Output Mix object to be used by player - no interfaces
required */
   res = (*EngineItf)->CreateOutputMix(EngineItf, &OutputMix, 0,
iidArray, required); CheckErr(res);

   /* Realizing the Output Mix object in synchronous mode. */
   res = (*OutputMix)->Realize(OutputMix, SL_BOOLEAN_FALSE);
CheckErr(res);

   /* Create 3D listener - no interfaces requires as the listener will
remain stationary */
   res = (*EngineItf)->CreateListener(EngineItf, &listener, 0, iidArray,
required); CheckErr(res);

   /* Realizing the listener object in synchronous mode. */
   res = (*listener)->Realize(listener, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Setup the data source structure for the player */
   uri.locatorType         = SL_DATALOCATOR_URI;
   uri.pURI                 = (SLchar *) "file:///buzzingbee.wav";
   mime.formatType         = SL_DATAFORMAT_MIME;
   mime.pMimeType           = (SLchar *) "audio/x-wav";
   mime.containerType      = SL_CONTAINERTYPE_WAV;

   audioSource.pLocator    = (void *)&uri;
   audioSource.pFormat     = (void *)&mime;
   /* Setup the data sink structure */
   locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
   locator_outputmix.outputMix     = OutputMix;
   audioSink.pLocator              = (void *)&locator_outputmix;
   audioSink.pFormat               = NULL;

   /* Set arrays required[] and iidArray[] for 3DLocationItf interface
(PlayItf is implicit) */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_3DLOCATION;

   /* Create the 3D player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, &player,
&audioSource, &audioSink, 1, iidArray, required); CheckErr(res);

   /* Realizing the player in synchronous mode. */
   res = (*player)->Realize(player, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Get the play and 3D location interfaces */
   res = (*player)->GetInterface(player, SL_IID_PLAY, (void*)&playItf);
CheckErr(res);
   res = (*player)->GetInterface(player, SL_IID_3DLOCATION,
(void*)&locationItf); CheckErr(res);

   {
      SLVec3D coords;
      SLuint32 playState;

      float angle = 0.0f;

      /* Position the 3D source in front the listener */
      coords.x = (SLuint32)( CIRCLE_RADIUS * sinf( angle ) );
      coords.y = 0;
      coords.z = (SLuint32)( CIRCLE_RADIUS * cosf( angle ) );
      (*locationItf)->SetLocationCartesian(locationItf, &coords);
CheckErr(res);

      /* Start playing the 3D source (buzzing bee) */
      res = (*playItf)->SetPlayState( playItf, SL_PLAYSTATE_PLAYING );
CheckErr(res);

      do
      {
         angle += CIRCLE_STEP;
         if( angle >= TWOPI )
         {
               angle = 0.0f;
         }

         /* move source in horizontal circle (clockwise direction) */
         coords.x = (SLuint32)( CIRCLE_RADIUS * sinf( angle ) );
         coords.z = (SLuint32)( CIRCLE_RADIUS * cosf( angle ) );
         (*locationItf)->SetLocationCartesian(locationItf, &coords);
CheckErr(res);

         SLEEP(10);  // Sleep for 10ms
         res = (*playItf)->GetPlayState(playItf, &playState);
CheckErr(res);
      } while ( playState != SL_PLAYSTATE_STOPPED );
   }

   /* Destroy the player */
   (*player)->Destroy(player);
   /* Destroy Output Mix object */
   (*OutputMix)->Destroy(OutputMix);
}

int sl_main( void )
{
   SLresult    res;
   SLObjectItf sl;

   SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };


   res = slCreateEngine( &sl, 3, EngineOption, 0, NULL, NULL);
CheckErr(res);
   /* Realizing the SL Engine in synchronous mode. */
   res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
   TestSimple3D(sl);
   /* Shutdown OpenSL ES */
   (*sl)->Destroy(sl);
   exit(0);
}
```

B.5.2 Advanced 3D

Simple 3D game showing use of priorities and advanced 3D properties.

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

#define SLEEP(x)        /* Client system sleep function to sleep x
milliseconds would replace SLEEP macro */

/* External game engine data */
```

```
#define EVENT_GUNSHOT   (int)0x00000001
#define EVENT_DEATH     (int)0x00000002
#define EVENT_FOOTSTEP  (int)0x00000003

#define OBJECT_LISTENER (int)0x00000001
#define OBJECT_GUNSHOT  (int)0x00000002
#define OBJECT_SCREAM   (int)0x00000003

/* External game engine functions */
extern int  GAMEGetEvents( void );
extern void GAMEGetLocation( int object, int *x, int *y, int *z);

/*******************************************************************/

#define MAX_NUMBER_INTERFACES 4

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      /* Debug printing to be placed here */
      exit(1);
   }
}

void Create3DSource( SLEngineItf EngineItf, SLObjectItf OutputMix,
SLObjectItf *pPlayer, SLchar *fileName, SLuint32 priority)
{
   SLDataSource            audioSource;
   SLDataLocator_URI       uri;
   SLDataFormat_MIME       mime;

   SLDataSink              audioSink;

   SLDataLocator_OutputMix locator_outputmix;

   SLresult                res;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   /* Setup the data source structure for the player */
   uri.locatorType         = SL_DATALOCATOR_URI;
   uri.pURI                 = fileName;
   mime.formatType         = SL_DATAFORMAT_MIME;
   mime.pMimeType           = (SLchar *) "audio/x-wav";
   mime.containerType      = SL_CONTAINERTYPE_WAV;
   audioSource.pLocator    = (void *)&uri;
   audioSource.pFormat     = (void *)&mime;

   /* Setup the data sink structure */
   locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
   locator_outputmix.outputMix     = OutputMix;
   audioSink.pLocator              = (void *)&locator_outputmix;
   audioSink.pFormat               = NULL;

   /* Set arrays required[] and iidArray[] for 3DLocationItf,
      3DSourceItf, 3DDopplerItf, SeekItf interfaces (PlayItf is
      implicit).
      Not all interfaces are used by all players in this example - in a
      real application it is advisable to only request interfaces that
      are necessary. */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_3DLOCATION;
   required[1] = SL_BOOLEAN_TRUE;
   iidArray[1] = SL_IID_3DSOURCE;
   required[2] = SL_BOOLEAN_FALSE;    /* Create the player even if
Doppler unavailable */
   iidArray[2] = SL_IID_3DDOPPLER;
   iidArray[3] = SL_IID_SEEK;
   required[3] = SL_BOOLEAN_TRUE;

   /* Create the 3D player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, pPlayer,
&audioSource, &audioSink, 4, iidArray, required); CheckErr(res);

   {
      SLObjectItf player = *pPlayer;

      /* Set player's priority */
      res = (*player)->SetPriority(player, priority, SL_BOOLEAN_TRUE);
CheckErr(res);

      /* Realize the player in synchronously */
      res = (*player)->Realize(player, SL_BOOLEAN_TRUE);
      if (res == SL_RESULT_RESOURCE_ERROR )
      {
         /* Ignore resource errors, they're handled elsewhere. */
      }
      else
      {
         CheckErr(res);
      }
   }
}

/* Play create a 3D source and spin it around the listener  */
void TestAdvanced3D( SLObjectItf sl )
{
   SLEngineItf              EngineItf;
   SL3DCommitItf            commitItf;

   SLresult                 res;
   SLObjectItf             gunshot, scream, footstep, torch;
   SLPlayItf               playItf;
   SL3DLocationItf         locationItf;
   SL3DSourceItf           sourceItf;
   SL3DDopplerItf          dopplerItf;

   SLObjectItf             listener;
   SLObjectItf             OutputMix;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   SLuint32                state;
   int i;

   /* Get the SL Engine Interface which is implicit */
   res = (*sl)->GetInterface(sl, SL_IID_ENGINE, (void*)&EngineItf);
CheckErr(res);

   /* Initialize arrays required[] and iidArray[] */
   for (i=0;i<MAX_NUMBER_INTERFACES;i++)
   {
      required[i] = SL_BOOLEAN_FALSE;
      iidArray[i] = SL_IID_NULL;
   }

   /* Get the commit interface and for efficiency reasons set into
deferred mode. */
   res = (*sl)->GetInterface(sl, SL_IID_3DCOMMIT, (void*)&commitItf);
CheckErr(res);
   (*commitItf)->SetDeferred(commitItf, SL_BOOLEAN_TRUE);

   /* Create Output Mix object to be used by player - no interfaces
required */
   res = (*EngineItf)->CreateOutputMix(EngineItf, &OutputMix, 0,
iidArray, required); CheckErr(res);

   /* Realizing the Output Mix object in synchronous mode. */
   res = (*OutputMix)->Realize(OutputMix, SL_BOOLEAN_FALSE);
CheckErr(res);

   /* Create 3D listener. */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_3DLOCATION;
   res = (*EngineItf)->CreateListener(EngineItf, &listener, 1, iidArray,
required); CheckErr(res);

   /* Realizing the listener object in synchronous mode. */
   res = (*listener)->Realize(listener, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Create four players with differing priorities. Higher priorities
      are used for sound effects that must be heard by the game player,
      whereas lower priorities are assigned to sound effects that make
      the game sound better but are not required to appreciate
      the game. */
   Create3DSource(EngineItf, OutputMix, &gunshot, "gunshot.wav",
SL_PRIORITY_HIGH);
   Create3DSource(EngineItf, OutputMix, &scream, "scream.wav",
SL_PRIORITY_NORMAL);
   Create3DSource(EngineItf, OutputMix, &footstep, "footstep.wav",
SL_PRIORITY_NORMAL);
   Create3DSource(EngineItf, OutputMix, &torch, "torch.wav",
SL_PRIORITY_LOW);

   (*gunshot)->GetState(gunshot, &state);
   if (state == SL_OBJECT_STATE_REALIZED)
   {
      /* Set the gun shot's 3D source properties */
      res = (*gunshot)->GetInterface(gunshot, SL_IID_3DSOURCE,
(void*)&sourceItf); CheckErr(res);
         /* Set rolloff model to linear */
      (*sourceItf)->SetRolloffModel(sourceItf, SL_ROLLOFFMODEL_LINEAR);
CheckErr(res);
         /* Exaggerate the gunshot's rolloff */
      (*sourceItf)->SetRolloffFactor(sourceItf, 1500); CheckErr(res);
         /* Add Doppler to the gunshot, if possible */
      res = (*gunshot)->GetInterface(gunshot, SL_IID_3DDOPPLER,
(void*)&dopplerItf);
      if (res != SL_RESULT_SUCCESS)
      {
         /* Doppler not available - not crucial though */
      }
      else
      {
         SLVec3D vec;
         /* Exaggerate gunshot's Doppler */
         (*dopplerItf)->SetDopplerFactor(dopplerItf, 2000);
CheckErr(res);
         /* Set gunshot's velocity to move away from the listener */
         vec.x = 0; vec.y = 0; vec.z = -1000;
         (*dopplerItf)->SetVelocityCartesian(dopplerItf, &vec);
CheckErr(res);
      }
   }
   else
   {
      /* Exit - game isn't viable without gunshot */
      exit(1);
   }

   (*footstep)->GetState(footstep, &state);
   if (state == SL_OBJECT_STATE_REALIZED)
   {
      /* Set foot step's 3D source properties */
      res = (*footstep)->GetInterface(footstep, SL_IID_3DSOURCE,
(void*)&sourceItf); CheckErr(res);
```

```
         /* Set foot steps as head relative - as the listener moves, so
            do the foot steps. */
      res = (*sourceItf)->SetHeadRelative(sourceItf, SL_BOOLEAN_TRUE);
CheckErr(res);
   }
   else
   {
      /* Exit - game isn't viable without gunshot */
      exit(1);
   }

   (*torch)->GetState(torch, &state);
   if (state == SL_OBJECT_STATE_REALIZED)
   {
      SLVec3D vec;
      SLSeekItf seekItf;

      res = (*torch)->GetInterface(torch, SL_IID_PLAY, (void*)&playItf);
CheckErr(res);
      res = (*torch)->GetInterface(torch, SL_IID_3DLOCATION, (void
*)&locationItf); CheckErr(res);
      res = (*torch)->GetInterface(torch, SL_IID_SEEK, (void*)&seekItf);
CheckErr(res);

      /* Position the torch somewhere in 3D space */
      vec.x = 30000; vec.y = 0; vec.z = -26000;
      (*locationItf)->SetLocationCartesian(locationItf, &vec);
CheckErr(res);

      /* Play torch constantly looping */
      (*seekItf)->SetLoop(seekItf, SL_BOOLEAN_TRUE, 0, SL_TIME_UNKNOWN);
CheckErr(res);
      /* Commit 3D settings before playing */
      (*commitItf)->Commit(commitItf);
      (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);
   }
   else
   {
      /* Torch isn't available. Could try realizing again but torch sound
         effect isn't crucial to game play. */
   }

   /* Main game loop */
   {
      int dead = 0;

      while (!dead)
      {
         int gameEvent;
         SLVec3D vec;

         /* Handle game events */
         gameEvent = GAMEGetEvents();
         switch( gameEvent )
         {
            case EVENT_GUNSHOT:
               /* Fire gun shot */
               res = (*gunshot)->GetInterface(gunshot, SL_IID_PLAY, (void
*)&playItf); CheckErr(res);
               (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);
               break;

            case EVENT_DEATH:
               /* Player has been shot, scream! */
               res = (*scream)->GetInterface(scream, SL_IID_PLAY, (void
*)&playItf); CheckErr(res);
               (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);
               dead = !dead;
               break;

            case EVENT_FOOTSTEP:
               /* Play foot steps */
               res = (*footstep)->GetInterface(footstep, SL_IID_PLAY,
(void *)&playItf); CheckErr(res);
               (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);
               break;
         }

         /* Update location of gun shot, scream and listener based on
            information from game engine. No need to update foot steps
            as they are head relative (i.e. move with the listener). */
         GAMEGetLocation(OBJECT_LISTENER, &vec.x, &vec.y, &vec.z);
         res = (*listener)->GetInterface(listener, SL_IID_3DLOCATION,
(void *)&locationItf); CheckErr(res);
         (*locationItf)->SetLocationCartesian(locationItf, &vec);
CheckErr(res);

         GAMEGetLocation(OBJECT_GUNSHOT, &vec.x, &vec.y, &vec.z);
         res = (*gunshot)->GetInterface(gunshot, SL_IID_3DLOCATION, (void
*)&locationItf); CheckErr(res);
         (*locationItf)->SetLocationCartesian(locationItf, &vec);
CheckErr(res);

         GAMEGetLocation(OBJECT_SCREAM, &vec.x, &vec.y, &vec.z);
         res = (*scream)->GetInterface(scream, SL_IID_3DLOCATION, (void
*)&locationItf); CheckErr(res);
         (*locationItf)->SetLocationCartesian(locationItf, &vec);
CheckErr(res);

         /* Commit 3D settings otherwise 3D positions will not be
updated. */
         (*commitItf)->Commit(commitItf);
         SLEEP(10);
      }
   }

   /* Wait until scream finished before exiting */
   (*scream)->GetState(scream, &state);
   if (state == SL_OBJECT_STATE_REALIZED)
   {
      res = (*scream)->GetInterface(scream, SL_IID_PLAY, (void
*)&playItf); CheckErr(res);
      do
      {
         (*playItf)->GetPlayState(playItf, &state); CheckErr(res);
         SLEEP(10);
      } while (state == SL_PLAYSTATE_PLAYING);
   }

   /* Destroy the players */
   (*gunshot)->Destroy(gunshot);
   (*scream)->Destroy(scream);
   (*footstep)->Destroy(footstep);
   (*torch)->Destroy(torch);

   /* Destroy the listener object */
   (*listener)->Destroy(listener);

   /* Destroy Output Mix object */
   (*OutputMix)->Destroy(OutputMix);
}

int sl_main( void )
{
   SLresult    res;
   SLObjectItf sl;

   SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };


   SLboolean required = SL_BOOLEAN_TRUE;
   SLInterfaceID iid = SL_IID_3DCOMMIT;

   /* Create an engine with the 3DCommit interface present */
   res = slCreateEngine( &sl, 3, EngineOption, 1, &iid, &required);
CheckErr(res);
   /* Realizing the SL Engine in synchronous mode. */
   res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
   TestAdvanced3D(sl);
```

```
    /* Shutdown OpenSL ES */
    (*sl)->Destroy(sl);
    exit(0);
}
```

B.6 Effects

B.6.1 Environmental Reverb

Creates a 3D scene with four sound sources constantly playing and background music. As the listener moves between rooms the reverb environment changes.

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

#define SLEEP(x)        // Client system sleep function to sleep x
milliseconds would replace SLEEP macro

/*******************************************************************/

#define MAX_NUMBER_INTERFACES 3

/* Four static objects */
#define STATIC0_FILENAME   "fireplace.wav"
#define STATIC0_RVB        0
#define STATIC1_FILENAME   "fountain.wav"
#define STATIC1_RVB        -300    /* Fountain has -3 dB reverb level */
#define STATIC2_FILENAME   "anvil.wav"
#define STATIC2_RVB        -300    /* Anvil has -3 dB reverb level */
#define STATIC3_FILENAME   "clocktick.wav"
#define STATIC3_RVB        -150    /* Clocktick has -1.5 dB reverb level
*/

/* Game engine objects */
#define OBJECT_LISTENER (int)0x00000001
#define OBJECT_STATIC0  (int)0x00000002
#define OBJECT_STATIC1  (int)0x00000003
#define OBJECT_STATIC2  (int)0x00000004
#define OBJECT_STATIC3  (int)0x00000005

/* Game engine rooms */
#define LOCATION_COURTYARD    (int)0x00000001
#define LOCATION_GREATHALL    (int)0x00000002
#define LOCATION_FRONTGARDEN  (int)0x00000003
#define LOCATION_BEDROOM      (int)0x00000004
#define LOCATION_EXIT         (int)0x00000005
#define LOCATION_UNKNOWN      (int)0xFFFFFFFF

/* External game engine functions */
extern void GAMEGetLocation( int object, int *x, int *y, int *z);
extern int GAMEGetListenerRoom( void );

/* Checks for error. If any errors exit the application! */
```

```
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      // Debug printing to be placed here
      exit(1);
   }
}

/* Create a 3D source positioned at (x, y, z), with reverb level
reverbLevel. */
void Create3DSource( SLEngineItf EngineItf, SLObjectItf OutputMix,
                     SLEnvironmentalReverbItf reverbItf,
                     SLObjectItf *pPlayer,
                     SLchar *fileName, SLint32 x, SLint32 y, SLint32 z,
                     SLmillibel reverbLevel)
{
   SLDataSource            audioSource;
   SLDataLocator_URI       uri;
   SLDataFormat_MIME       mime;

   SLDataSink              audioSink;

   SLDataLocator_OutputMix locator_outputmix;

   SL3DLocationItf         locationItf;
   SLEffectSendItf         effectSendItf;
   SLSeekItf               seekItf;
   SLPlayItf               playItf;

   SLresult                res;

   SLVec3D                 coords;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   /* Setup the data source structure for the player */
   uri.locatorType         = SL_DATALOCATOR_URI;
   uri.pURI                 = fileName;
   mime.formatType         = SL_DATAFORMAT_MIME;
   mime.pMimeType           = (SLchar *) "audio/x-wav";
   mime.containerType      = SL_CONTAINERTYPE_WAV;
   audioSource.pLocator    = (void *)&uri;
   audioSource.pFormat     = (void *)&mime;

   /* Setup the data sink structure */
   locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
   locator_outputmix.outputMix     = OutputMix;
   audioSink.pLocator              = (void *)&locator_outputmix;
   audioSink.pFormat               = NULL;
   /* Set arrays required[] and iidArray[] for 3DLocationItf,
EffectSendItf, SeekItf interfaces (PlayItf is implicit). */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_3DLOCATION;
   required[1] = SL_BOOLEAN_TRUE;
   iidArray[1] = SL_IID_EFFECTSEND;
   required[2] = SL_BOOLEAN_TRUE;
   iidArray[2] = SL_IID_SEEK;

   /* Create the 3D player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, pPlayer,
&audioSource, &audioSink, 3, iidArray, required); CheckErr(res);

   {
      SLObjectItf player   = *pPlayer;

      /* Realize the player in synchronously */
      res = (*player)->Realize(player, SL_BOOLEAN_TRUE); CheckErr(res);

      /* Get the 3D location interfaces, set the 3D position */
      res = (*player)->GetInterface(player, SL_IID_3DLOCATION, (void
*)&locationItf); CheckErr(res);
      coords.x = x;
      coords.y = y;
      coords.z = z;
      res = (*locationItf)->SetLocationCartesian(locationItf, &coords);

      /* Get the effect send interface, set the reverb level for the
sound source */
      res = (*player)->GetInterface(player, SL_IID_EFFECTSEND, (void
*)&effectSendItf); CheckErr(res);
      (*effectSendItf)->EnableEffectSend(effectSendItf, reverbItf,
SL_BOOLEAN_TRUE, reverbLevel);

      /* Get the seek interface and enable looping of the whole file */
      (*player)->GetInterface(player, SL_IID_SEEK, (void *)&seekItf);
CheckErr(res);
      (*seekItf)->SetLoop(seekItf, SL_BOOLEAN_TRUE, 0, SL_TIME_UNKNOWN);
CheckErr(res);

      /* Get the play interface and start playing. */
      res = (*player)->GetInterface(player, SL_IID_PLAY, (void
*)&playItf); CheckErr(res);
      (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);
   }
}

/* A listener moves around a scene that contains four
   sound sources. As the listener moves the reverb
   changes to adapt to the environment the listener is in.
   Music is also played in the background.                  */
void TestEnvironmentalReverb( SLObjectItf sl )
```

```
{
   SLEngineItf             EngineItf;

   SLresult                res;

   SLDataSource            audioSource;
   SLDataLocator_URI       uri;
   SLDataFormat_MIME       mime;

   SLDataSink              audioSink;
   SLDataLocator_OutputMix locator_outputmix;

   SLObjectItf             music;
   SLPlayItf               playItf;

   SLObjectItf             player[4];

   SLObjectItf             listener;
   SL3DLocationItf         listenerLocationItf;

   SLObjectItf             OutputMix;
   SLEnvironmentalReverbItf reverbItf;

   int i;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   /* Get the SL Engine Interface which is implicit */
   res = (*sl)->GetInterface(sl, SL_IID_ENGINE, (void *)&EngineItf);
CheckErr(res);

   /* Initialize arrays required[] and iidArray[] */
   for (i=0;i<MAX_NUMBER_INTERFACES;i++)
   {
      required[i] = SL_BOOLEAN_FALSE;
      iidArray[i] = SL_IID_NULL;
   }

   /**** OUTPUT MIX **************************************/
   /* Uses the default output device(s).                 */
   /* Includes environmental reverb auxiliary effect for */
   /* players requiring use of reverb (in this case the  */
   /* anvil).                                            */
   /*****************************************************/

   /* Create Output Mix object to be used by player,
      requesting use of the environmental reverb
      interface. */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_ENVIRONMENTALREVERB;
   res = (*EngineItf)->CreateOutputMix(EngineItf, &OutputMix, 1,
iidArray, required); CheckErr(res);
```

```
   /* Realizing the Output Mix object in synchronous mode. */
   res = (*OutputMix)->Realize(OutputMix, SL_BOOLEAN_FALSE);
CheckErr(res);

   /* Get environment reverb interface. */
   res = (*OutputMix)->GetInterface(OutputMix,
SL_IID_ENVIRONMENTALREVERB, (void *)&reverbItf); CheckErr(res);

   /**** LISTENER ****************************************/
   /* Listener that we'll move around in response to     */
   /* application events (e.g. d-pad movement).          */
   /******************************************************/

   /* Create 3D listener */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_3DLOCATION;
   res = (*EngineItf)->CreateListener(EngineItf, &listener, 1, iidArray,
required); CheckErr(res);

   /* Realizing the listener object in synchronous mode. */
   res = (*listener)->Realize(listener, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Get listener's location interface. */
   res = (*listener)->GetInterface(listener, SL_IID_3DLOCATION, (void
*)&listenerLocationItf); CheckErr(res);

   /**** BACKGROUND MUSIC ********************************/
   /* Music that's played in the background.             */
   /******************************************************/

   /* Setup the data source structure for the background music */
   uri.locatorType         = SL_DATALOCATOR_URI;
   uri.pURI                 = (SLchar *) "file:///backgroundmusic.wav";
   mime.formatType         = SL_DATAFORMAT_MIME;
   mime.pMimeType           = (SLchar *) "audio/x-wav";
   mime.containerType      = SL_CONTAINERTYPE_WAV;
   audioSource.pLocator    = (void *)&uri;
   audioSource.pFormat     = (void *)&mime;

   /* Setup the data sink structure */
   locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
   locator_outputmix.outputMix     = OutputMix;
   audioSink.pLocator              = (void *)&locator_outputmix;
   audioSink.pFormat               = NULL;

   /* The background music should be rendered in 2D so we do not
      request the 3DLocationItf interface.
      We also do not want the music to have reverb applied to it so
      we do not request the EffectSendItf interfaces (PlayItf is
implicit). */

   /* Create the music player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, &music, &audioSource,
&audioSink, 0, iidArray, required); CheckErr(res);

   /* Realizing the player in synchronous mode. */
   res = (*music)->Realize(music, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Start playing music */
   res = (*music)->GetInterface(music, SL_IID_PLAY, (void *)&playItf);
CheckErr(res);
   (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING);
CheckErr(res);

   /**** OBJECTS *****************************************/
   /* There are four looping sound sources positioned in */
   /* various places in the 3D scene (as determined by   */
   /* the game engine), each included in the             */
   /* environmental reverb.                              */
   /******************************************************/
   {
      SLint32 x, y, z;

      GAMEGetLocation(OBJECT_STATIC0, &x, &y, &z);
      Create3DSource(EngineItf, OutputMix, reverbItf, &player[0],
STATIC0_FILENAME, x, y, z, STATIC0_RVB);

      GAMEGetLocation(OBJECT_STATIC1, &x, &y, &z);
      Create3DSource(EngineItf, OutputMix, reverbItf, &player[1],
STATIC1_FILENAME, x, y, z, STATIC1_RVB);

      GAMEGetLocation(OBJECT_STATIC2, &x, &y, &z);
      Create3DSource(EngineItf, OutputMix, reverbItf, &player[2],
STATIC2_FILENAME, x, y, z, STATIC2_RVB);

      GAMEGetLocation(OBJECT_STATIC3, &x, &y, &z);
      Create3DSource(EngineItf, OutputMix, reverbItf, &player[3],
STATIC3_FILENAME, x, y, z, STATIC3_RVB);
   }

   /* Main loop */
   {
      int exit = 0;
      int oldEnvironment = LOCATION_UNKNOWN;

      while (!exit)
      {
         SLVec3D vec;
         int environment;

         /* Update location listener based on information from game
engine. */
         GAMEGetLocation(OBJECT_LISTENER, &vec.x, &vec.y, &vec.z);
         (*listenerLocationItf)-
>SetLocationCartesian(listenerLocationItf, &vec); CheckErr(res);
```

```
         /* Change the listener's environment based on the room the
listener is located in. */
         environment = GAMEGetListenerRoom();
         if (environment != oldEnvironment)
         {
            switch (environment)
            {
            case LOCATION_COURTYARD:
               {
                  SLEnvironmentalReverbSettings rvbSettings =
SL_I3DL2_ENVIRONMENT_PRESET_QUARRY;
                  (*reverbItf)-
>SetEnvironmentalReverbProperties(reverbItf, &rvbSettings);
               }
               break;
            case LOCATION_GREATHALL:
               {
                  SLEnvironmentalReverbSettings rvbSettings =
SL_I3DL2_ENVIRONMENT_PRESET_LARGEHALL;
                  (*reverbItf)-
>SetEnvironmentalReverbProperties(reverbItf, &rvbSettings);
               }
               break;
            case LOCATION_FRONTGARDEN:
               {
                  SLEnvironmentalReverbSettings rvbSettings =
SL_I3DL2_ENVIRONMENT_PRESET_GENERIC;
                  (*reverbItf)-
>SetEnvironmentalReverbProperties(reverbItf, &rvbSettings);
               }
               break;
            case LOCATION_BEDROOM:
               {
                  SLEnvironmentalReverbSettings rvbSettings =
SL_I3DL2_ENVIRONMENT_PRESET_SMALLROOM;
                  (*reverbItf)-
>SetEnvironmentalReverbProperties(reverbItf, &rvbSettings);
               }
               break;
            case LOCATION_EXIT:
               exit = 1;
               break;
            }
            oldEnvironment = environment;
         }

         SLEEP(10);
      }
   }

   /* Destroy the players */
   (*music)->Destroy(music);
   (*player[0])->Destroy(player[0]);
   (*player[1])->Destroy(player[1]);
   (*player[2])->Destroy(player[2]);
   (*player[3])->Destroy(player[3]);

   /* Destroy the listener object */
   (*listener)->Destroy(listener);

   /* Destroy Output Mix object */
   (*OutputMix)->Destroy(OutputMix);
}

int sl_main( void )
{
   SLresult    res;
   SLObjectItf sl;

   SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };


   res = slCreateEngine( &sl, 3, EngineOption, 0, NULL, NULL);
CheckErr(res);
   /* Realizing the SL Engine in synchronous mode. */
   res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
   TestEnvironmentalReverb(sl);
   /* Shutdown OpenSL ES */
   (*sl)->Destroy(sl);
   exit(0);
}
```

B.6.2 Equalizer

This example shows the OpenSL ES part of an interactive equalizer GUI.

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>


#define MAX_NUMBER_INTERFACES 5

/* Global variables. (Should be local in real application.) */
SLObjectItf         engine; /* OpenSL ES Engine */
SLObjectItf         player;
SLObjectItf         outputMix;
SLPlayItf           playItf;
```

```
SLEqualizerItf      equalizerItf;

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
     // Debug printing to be placed here
     exit(1);
   }
}

/*
 * Draws single EQ band to the screen. Called by drawEQDisplay
 */
void drawEQBand(int minFreq, int maxFreq, int level)
{
   /* insert drawing routines here for single EQ band
     (use GetBandLevelRange and screen height to map the level to screen
y-coordinate) */
}

/*
 * Called when the display is repainted.
 */
void drawEQDisplay()
{
   SLuint16 numBands;
   SLmillibel bandLevel, minLevel, maxLevel;
   SLmilliHertz minFreq, maxFreq;
   int band;

   SLresult res;

   res = (*equalizerItf)->GetNumberOfBands( equalizerItf, &numBands );
CheckErr(res);
   res = (*equalizerItf)->GetBandLevelRange( equalizerItf, &minLevel,
&maxLevel ); CheckErr(res);

   for(band = 0; band<numBands; band++)
   {
      res = (*equalizerItf)->GetBandFreqRange( equalizerItf,
(SLint16)band, &minFreq, &maxFreq ); CheckErr(res);
      res = (*equalizerItf)->GetBandLevel( equalizerItf, (SLint16)band,
&bandLevel ); CheckErr(res);
      drawEQBand(minFreq, maxFreq, bandLevel);
   }
}

/*
 * Initializes the OpenSL ES engine and start the playback of
 * some music from a file and draw the graphical equalizer
 */
```

```
void init()
{
   SLEngineItf             EngineItf;

   SLresult                res;

   SLDataSource            audioSource;
   SLDataLocator_URI       uri;
   SLDataFormat_MIME       mime;

   SLDataSink              audioSink;
   SLDataLocator_OutputMix locator_outputmix;

   SLVolumeItf             volumeItf;

   int                     i;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };



   /* Create OpenSL ES */
   res = slCreateEngine( &engine, 3, EngineOption, 0, NULL, NULL);
CheckErr(res);
   /* Realizing the SL Engine in synchronous mode. */
   res = (*engine)->Realize(engine, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Get the SL Engine Interface which is implicit*/
   res = (*engine)->GetInterface(engine, SL_IID_ENGINE, (void
*)&EngineItf); CheckErr(res);

   /* Initialize arrays required[] and iidArray[] */
   for (i=0;i<MAX_NUMBER_INTERFACES;i++)
   {
      required[i] = SL_BOOLEAN_FALSE;
      iidArray[i] = SL_IID_NULL;
   }

   /* Set arrays required[] and iidArray[] for VOLUME and EQUALIZER
interfaces */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_VOLUME;
   required[1] = SL_BOOLEAN_TRUE;
   iidArray[1] = SL_IID_EQUALIZER;
   /* Create Output Mix object to be used by player */
   res = (*EngineItf)->CreateOutputMix(EngineItf, &outputMix, 2,
iidArray, required); CheckErr(res);

   /* Realizing the Output Mix object in synchronous mode. */
   res = (*outputMix)->Realize(outputMix, SL_BOOLEAN_FALSE);
CheckErr(res);

   /* Get play and equalizer interface */
   res = (*outputMix)->GetInterface(outputMix, SL_IID_VOLUME, (void
*)&volumeItf); CheckErr(res);
   res = (*outputMix)->GetInterface(outputMix, SL_IID_EQUALIZER, (void
*)&equalizerItf); CheckErr(res);

   /* Setup the data source structure */
   uri.locatorType         = SL_DATALOCATOR_URI;
   uri.pURI                 = (SLchar *) "file:///music.wav";
   mime.formatType         = SL_DATAFORMAT_MIME;
   mime.pMimeType           = (SLchar *) "audio/x-wav";
   mime.containerType      = SL_CONTAINERTYPE_WAV;

   audioSource.pLocator    = (void *)&uri;
   audioSource.pFormat     = (void *)&mime;

   /* Setup the data sink structure */
   locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
   locator_outputmix.outputMix     = outputMix;
   audioSink.pLocator              = (void *)&locator_outputmix;
   audioSink.pFormat               = NULL;

   /* Set arrays required[] and iidArray[] for no interfaces (PlayItf is
implicit) */
   required[0] = SL_BOOLEAN_FALSE;
   iidArray[0] = SL_IID_NULL;
   required[1] = SL_BOOLEAN_FALSE;
   iidArray[1] = SL_IID_NULL;

   /* Create the music player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, &player,
&audioSource, &audioSink, 0, iidArray, required); CheckErr(res);

   /* Realizing the player in synchronous mode. */
   res = (*player)->Realize(player, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Get the play interface */
   res = (*player)->GetInterface(player, SL_IID_PLAY, (void *)&playItf);
CheckErr(res);

   /* Before we start set volume to -3dB (-300mB) and enable equalizer */
   res = (*volumeItf)->SetVolumeLevel(volumeItf, -300); CheckErr(res);
   res = (*equalizerItf)->SetEnabled(equalizerItf, SL_BOOLEAN_TRUE);
CheckErr(res);
```

```
   /* Play the music */
   res = (*playItf)->SetPlayState( playItf, SL_PLAYSTATE_PLAYING );
CheckErr(res);

   /* Draw the graphical EQ */
   drawEQDisplay();
}

/**
 * Shuts down the OpenSL ES engine.
 */
void  destroy()
{
   SLresult    res;

   /* Stop the music */
   res = (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_STOPPED);
CheckErr(res);
   /* Destroy the player */
   (*player)->Destroy(player);
   /* Destroy Output Mix object */
   (*outputMix)->Destroy(outputMix);

   /* Shutdown OpenSL ES */
   (*engine)->Destroy(engine);
}

/*
 * Called by UI when user increases or decreases a band level.
 */
void setBandLevel(SLint16 band, SLboolean increase)
{
   SLuint16 numBands;
   SLmillibel bandLevel, minLevel, maxLevel;

   SLresult res;

   res = (*equalizerItf)->GetNumberOfBands( equalizerItf, &numBands );
CheckErr(res);
   res = (*equalizerItf)->GetBandLevelRange( equalizerItf, &minLevel,
&maxLevel ); CheckErr(res);

   if( band >= numBands ) {
      /* Error. Insert debug print here. */
      exit(0);
   }

   res = (*equalizerItf)->GetBandLevel( equalizerItf, band, &bandLevel );
CheckErr(res);

   if( increase==SL_BOOLEAN_TRUE )
   {
      /* increase the level by 1 dB (100mB) if the max supported level is
not exceeded */
      bandLevel = bandLevel + 100;
      if( bandLevel < maxLevel )
      {
         res = (*equalizerItf)->SetBandLevel( equalizerItf, band,
bandLevel ); CheckErr(res);
         drawEQDisplay();
      }
   } else /* increase==false */
   {
      /* decrease the level by 1 dB (100mB) if the min supported level is
not crossed */
      bandLevel = bandLevel - 100;
      if( bandLevel > minLevel )
      {
         res = (*equalizerItf)->SetBandLevel( equalizerItf, band,
bandLevel ); CheckErr(res);
         drawEQDisplay();
      }
   }
}
```

B.7 IO Devices and capabilities

B.7.1 Engine capabilities

Engine Capabilities Example.

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

#define MAX_NUMBER_LED_DEVICES 3
#define MAX_NUMBER_VIBRA_DEVICES 3
#define POSITION_UPDATE_PERIOD 1000 /* 1 sec */

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      /* Debug printing to be placed here */
      exit(1);
   }
}

/*
 * Test the querying of capabilities of an OpenSL ES engine
 */
void TestEngineCapabilities(SLObjectItf sl)
{
   SLEngineCapabilitiesItf EngineCapabilitiesItf;
   SLVibraDescriptor VibraDescriptor[MAX_NUMBER_VIBRA_DEVICES];
   SLLEDDescriptor LEDDescriptor[MAX_NUMBER_LED_DEVICES];

   SLresult                res;

   SLuint32 i = 0, numLEDDevices = 0,
LEDDeviceID[MAX_NUMBER_LED_DEVICES];
   SLuint32 numVibraDevices = 0, VibraDeviceID[MAX_NUMBER_VIBRA_DEVICES];
   SLboolean isThreadSafe = SL_BOOLEAN_FALSE;

   SLint16 profilesSupported = 0;
   SLboolean isPhoneProfileSupported = SL_BOOLEAN_FALSE;
   SLboolean isMusicProfileSupported = SL_BOOLEAN_FALSE;
   SLboolean isGameProfileSupported = SL_BOOLEAN_FALSE;
   SLint16 numMIDISynthesizers = 0;

   SLint16 numMax2DVoices = 0, numFree2DVoices = 0;
   SLboolean isAbsoluteMax2D = SL_BOOLEAN_FALSE;
   SLint16 numMaxMIDIVoices = 0, numFreeMIDIVoices = 0;
   SLboolean isAbsoluteMaxMIDI = SL_BOOLEAN_FALSE;

   SLint16 numMax3DVoices = 0, numFree3DVoices = 0;
   SLboolean isAbsoluteMax3D = SL_BOOLEAN_FALSE;

   SLint16 numMax3DMidiVoices = 0, numFree3DMidiVoices = 0;
   SLboolean isAbsoluteMax3DMidi = SL_BOOLEAN_FALSE;

   SLint16 vMajor = 0, vMinor = 0, vStep = 0;

   /* Get the Engine Capabilities interface - an implicit interface */
   res = (*sl)->GetInterface(sl, SL_IID_ENGINECAPABILITIES, (void
*)&EngineCapabilitiesItf); CheckErr(res);

   /* Query profile support */
   res = (*EngineCapabilitiesItf)->QuerySupportedProfiles(
EngineCapabilitiesItf, &profilesSupported); CheckErr(res);

   if (profilesSupported & SL_PROFILES_PHONE)
      isPhoneProfileSupported = SL_BOOLEAN_TRUE;
   if (profilesSupported & SL_PROFILES_MUSIC)
      isMusicProfileSupported = SL_BOOLEAN_TRUE;
   if (profilesSupported & SL_PROFILES_GAME)
      isGameProfileSupported = SL_BOOLEAN_TRUE;

   /* Query available voices for 2D audio */
   res = (*EngineCapabilitiesItf)->QueryAvailableVoices(
EngineCapabilitiesItf, SL_VOICETYPE_2D_AUDIO, &numMax2DVoices,
&isAbsoluteMax2D, &numFree2DVoices); CheckErr(res);

   /* Query available voices for MIDI. Note: MIDI is mandated only in the
PHONE and GAME profiles. */
   res = (*EngineCapabilitiesItf)->QueryAvailableVoices(
EngineCapabilitiesItf, SL_VOICETYPE_MIDI, &numMaxMIDIVoices,
&isAbsoluteMaxMIDI, &numFreeMIDIVoices); CheckErr(res);

   /* 3D audio functionality is mandated only in the game profile, so
might want to query for 3D voice types only if GAME profile is supported
*/
   if (isGameProfileSupported) {
      res = (*EngineCapabilitiesItf)->QueryAvailableVoices(
EngineCapabilitiesItf, SL_VOICETYPE_3D_AUDIO, &numMax3DVoices,
&isAbsoluteMax3D, &numFree3DVoices); CheckErr(res);
      res = (*EngineCapabilitiesItf)->QueryAvailableVoices(
EngineCapabilitiesItf, SL_VOICETYPE_3D_MIDIOUTPUT, &numMax3DMidiVoices,
&isAbsoluteMax3DMidi, &numFree3DMidiVoices); CheckErr(res);
   }

   /* Query number of MIDI synthesizers */
   res = (*EngineCapabilitiesItf)->QueryNumberOfMIDISynthesizers(
EngineCapabilitiesItf, &numMIDISynthesizers); CheckErr(res);
```

```
   /* Do something with MIDI synthesizer information */

   /* Query API version */
   res = (*EngineCapabilitiesItf)->QueryAPIVersion(
EngineCapabilitiesItf, &vMajor, &vMinor, &vStep); CheckErr(res);

   /* Do something with API version information */


      /* Query number of LED devices present in the system */
   res = (*EngineCapabilitiesItf)->QueryLEDCapabilities(
EngineCapabilitiesItf, &numLEDDevices, NULL, NULL); CheckErr(res);

   /* Get the capabilities of each LED device present */
   for(i=0; i< numLEDDevices; i++) {
   /* Retrieve the LEDdeviceID for each of the LED devices found on the
system */
      res = (*EngineCapabilitiesItf)->QueryLEDCapabilities(
EngineCapabilitiesItf, &i, &LEDDeviceID[i], NULL); CheckErr(res);
   /* Either the index i or the LEDdeviceID can be used to retrieve the
capabilities of each LED device; we choose to use the LEDDeviceID here */
      res = (*EngineCapabilitiesItf)->QueryLEDCapabilities(
EngineCapabilitiesItf, NULL, &LEDDeviceID[i], &LEDDescriptor[i]);
      CheckErr(res);
   }

   /* Query number of vibra devices present in the system */
   res = (*EngineCapabilitiesItf)->QueryVibraCapabilities(
EngineCapabilitiesItf, &numVibraDevices, NULL, NULL); CheckErr(res);

   /* Get the capabilities of each vibra device present */
   for(i=0;i< numVibraDevices; i++) {
   /* Retrieve the VibradeviceID for each of the Vibra devices found on
the system */
      res = (*EngineCapabilitiesItf)->QueryVibraCapabilities(
EngineCapabilitiesItf, &i, &VibraDeviceID[i], NULL); CheckErr(res);
   /* Either the index i or the VibraDeviceID can be used to retrieve the
capabilities of each Vibra device; we choose to use the VibraDeviceID
here */
      res = (*EngineCapabilitiesItf)->QueryVibraCapabilities(
EngineCapabilitiesItf, NULL, &VibraDeviceID[i], &VibraDescriptor[i]);
CheckErr(res);
   }

   /* Determine if implementation is thread safe */
   res = (*EngineCapabilitiesItf)->IsThreadSafe( EngineCapabilitiesItf,
&isThreadSafe); CheckErr(res);

   /* Do something with thread-safety information returned */
}

int sl_main( void )
{
   SLresult    res;
   SLObjectItf sl;

   /* Create OpenSL ES engine in thread-safe mode */
   SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };


   res = slCreateEngine( &sl, 3, EngineOption, 0, NULL, NULL);
CheckErr(res);

   /* Realizing the SL Engine in synchronous mode. */
   res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
   TestEngineCapabilities(sl);
   /* Shutdown OpenSL ES */
   (*sl)->Destroy(sl);
   exit(0);
}
```

Appendix C: Use Case Sample Code

C.1 Introduction

This appendix provides sample code illustrating how objects can be used together to support one simple use case for each profile. A description of each use case is followed by an object relationship diagram and sample code. The sample code shows how to use the API and is for purposes of illustration only – it is not intended to provide realistic application code. Specifically, the sample code uses getchar for purposes of illustration only and includes limited error handling for the sake of clarity.

C.2 Music Profile

This example shows how the API is used in the following case:

Play some music from a file, adding/removing Reverb and enabling/disabling Stereo Positioning when requested by a user. The application also receives music playback progress indications via callbacks, so that a progress bar can be displayed for the user. The resources for Reverb are added and removed dynamically to illustrate use of the Dynamic Interface Management functionality.

C.2.1 Object Relationships

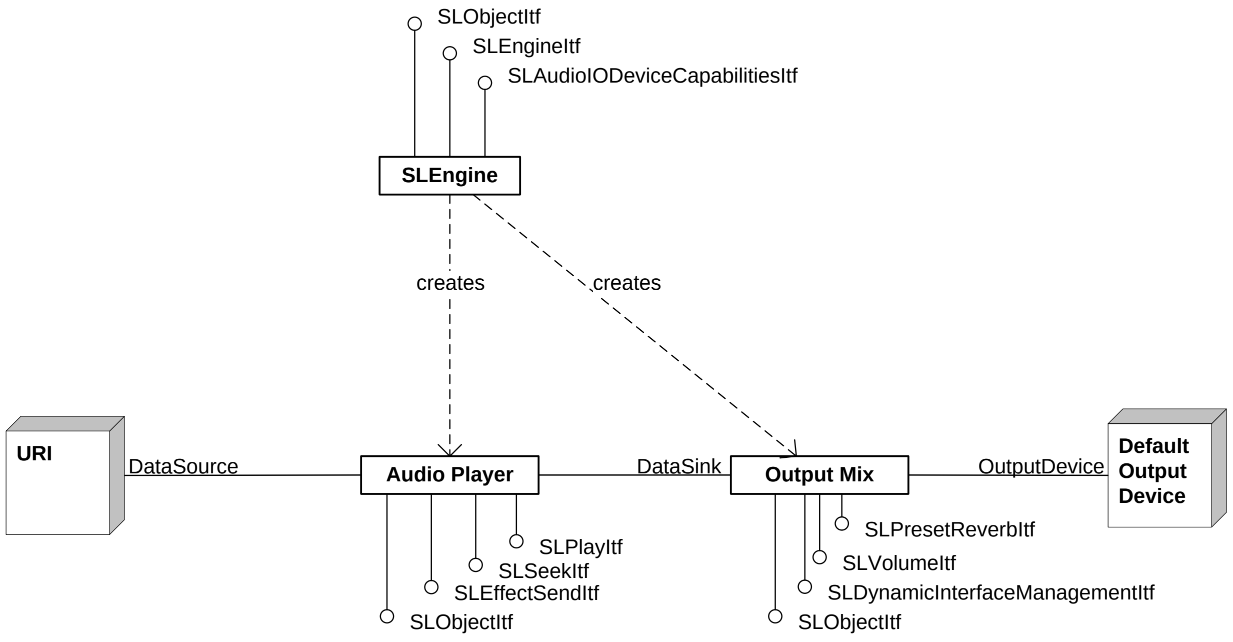


Figure 42: Object relationships

C.2.2 Example Code

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

#define MAX_NUMBER_INTERFACES 4
#define MAX_NUMBER_OUTPUT_DEVICES 6
#define POSITION_UPDATE_PERIOD 1000     /* 1 second */

/*******************************************************************/
/* Dummy semaphore and event related types, prototypes and defines */

typedef SLuint16 Sem_t;  /* System semaphore type would replace Sem_t */
Sem_t semDIM;

void sem_post(Sem_t *pSemaphore)
{
   /* Implementation specific semaphore post */
}
void sem_wait(Sem_t *pSemaphore)
{
   /* Implementation specific semaphore wait */
}

#define MAX_NUMBER_EVENTS 50
void* eventQueue[MAX_NUMBER_EVENTS];

void event_post(void* eventQueue[], void* event)
{
   /* Implementation specific event post */
};

void* event_read(void* eventQueue[])
{
   void* result_p = NULL;
   /* Implementation specific event read */
   return result_p;
};

/*******************************************************************/

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      /* Debug printing to be placed here */
      exit(1);
   }
}
```

```
SLresult  asyncRes;    /* Global variable used to pass result back to
client */

void DIMCallback(    SLDynamicInterfaceManagementItf  caller,
                           void                             *pContext,
                           SLuint32                         event,
                           SLresult                         result,
                           const SLInterfaceID              iid)
{
   if(event==SL_DYNAMIC_ITF_EVENT_ASYNC_TERMINATION)
   {
      if((iid == SL_IID_PRESETREVERB))
      {
         asyncRes=result;   /* Better and safer to set member of
*pContext in context of client thread
                             *        but global variable used to pass
result back to client for simplicity */
         sem_post(&semDIM);
      }
   }
   else
   {
      /* Debug printing to be placed here */
      exit(1);
   }
}

typedef struct queue_playevent_t_ {
   SLPlayItf caller;
   void      *pContext;
   SLuint32  playevent;
} queue_playevent_t;

/* Callback for position events */
void PlayEventCallback(SLPlayItf caller,
                                  void      *pContext,
                                  SLuint32  playevent)
{
   queue_playevent_t queue_playevent={caller, pContext, playevent};
   event_post(eventQueue, (void*)&queue_playevent);
}

/* Example main event loop thread for handling Play events*/
void eventThread()
{
   queue_playevent_t* queue_playevent_p;
   while(1) /* Event loop */
   {
      queue_playevent_p=(queue_playevent_t*)event_read(eventQueue);
      switch(queue_playevent_p->playevent)
      {
         case SL_PLAYEVENT_HEADATNEWPOS:
         /* Advance progress bar by 1 second */
```

```
            break;
         default:
            break;
      }
   }
}

/* Play some music from a file, adding PresetReverb and changing stereo
position when requested as well as doing progress bar callbacks */
void TestPlayMusicStereoPositionAndReverb( SLObjectItf sl )
{
   SLEngineItf                      EngineItf;
   SLAudioIODeviceCapabilitiesItf  AudioIODeviceCapabilitiesItf;
   SLAudioOutputDescriptor         AudioOutputDescriptor;

   SLuint32 OutputDeviceIDs[MAX_NUMBER_OUTPUT_DEVICES];
   SLint32   numOutputs = 0;
   SLboolean earpiece_available = SL_BOOLEAN_FALSE;
   SLboolean handsfree_speaker_available = SL_BOOLEAN_FALSE;
   SLboolean earpiece_or_handsfree_speaker_default = SL_BOOLEAN_FALSE;
   SLuint32 earpiece_deviceID = 0;
   SLuint32 handsfree_speaker_deviceID = 0;
   SLuint32 deviceID = 0;

   SLresult                res;

   SLDataSource            audioSource;
   SLDataLocator_URI       uri;
   SLDataFormat_MIME       mime;

   SLDataSink              audioSink;
   SLDataLocator_OutputMix locator_outputmix;

   SLObjectItf             player;
   SLPlayItf               playItf;
   SLSeekItf               seekItf;
   SLEffectSendItf         effectSendItf;

   SLObjectItf             OutputMix;
   SLVolumeItf             volumeItf;
   SLDynamicInterfaceManagementItf dynamicInterfaceManagementItf;

   int                     i;
   char                    c;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   /*Get the SL Engine Interface which is implicit*/
   res = (*sl)->GetInterface(sl, SL_IID_ENGINE, (void *)&EngineItf);
CheckErr(res);

   /* Initialize arrays required[] and iidArray[] */
   for (i=0;i<MAX_NUMBER_INTERFACES;i++)
   {
      required[i] = SL_BOOLEAN_FALSE;
      iidArray[i] = SL_IID_NULL;
   }

   /* Get the Audio IO DEVICE CAPABILITIES interface */
   res = (*sl)->GetInterface(sl, SL_IID_AUDIOIODEVICECAPABILITIES, (void
*)&AudioIODeviceCapabilitiesItf); CheckErr(res);
   numOutputs = MAX_NUMBER_OUTPUT_DEVICES;
   res = (*AudioIODeviceCapabilitiesItf)->GetAvailableAudioOutputs(
AudioIODeviceCapabilitiesItf, &numOutputs, OutputDeviceIDs);
CheckErr(res);

   /* Search for phone earpiece output and phone speaker device */
   for (i=0;i<numOutputs; i++)
   {
      res = (*AudioIODeviceCapabilitiesItf)-
>QueryAudioOutputCapabilities(AudioIODeviceCapabilitiesItf,
OutputDeviceIDs[i], &AudioOutputDescriptor); CheckErr(res);
      if((AudioOutputDescriptor.deviceConnection ==
SL_DEVCONNECTION_INTEGRATED)&&
          (AudioOutputDescriptor.deviceScope == SL_DEVSCOPE_USER)&&
          (AudioOutputDescriptor.deviceLocation ==
SL_DEVLOCATION_HANDSET))
      {
         earpiece_deviceID = OutputDeviceIDs[i];
         earpiece_available = SL_BOOLEAN_TRUE;
      }
      else if((AudioOutputDescriptor.deviceConnection ==
SL_DEVCONNECTION_INTEGRATED)&&
                  (AudioOutputDescriptor.deviceScope ==
SL_DEVSCOPE_ENVIRONMENT)&&
                  (AudioOutputDescriptor.deviceLocation ==
SL_DEVLOCATION_HANDSET))
      {
         handsfree_speaker_deviceID = OutputDeviceIDs[i];
         handsfree_speaker_available = SL_BOOLEAN_TRUE;
      }
   }

   numOutputs = MAX_NUMBER_OUTPUT_DEVICES;
   res = (*AudioIODeviceCapabilitiesItf)-
>GetDefaultAudioDevices(AudioIODeviceCapabilitiesItf,
SL_DEFAULTDEVICEID_AUDIOOUTPUT, &numOutputs, OutputDeviceIDs);
CheckErr(res);
   /* Check whether Default Output devices include either earpiece or
phone speaker */
   for (i=0;i<numOutputs; i++)
   {
      if((OutputDeviceIDs[i] == earpiece_deviceID)||
          (OutputDeviceIDs[i] == handsfree_speaker_deviceID))
      {
         earpiece_or_handsfree_speaker_default = SL_BOOLEAN_TRUE;
         break;
      }
   }

   /* Expect earpiece or phone speaker to be set as default output device
*/
   if(!earpiece_or_handsfree_speaker_default)
   {
      /* Debug printing to be placed here */
      exit(1);
   }

   /* Set arrays required[] and iidArray[] for VOLUME interface */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_VOLUME;
   /* Create Output Mix object to be used by player */
   res = (*EngineItf)->CreateOutputMix(EngineItf, &OutputMix, 1,
iidArray, required); CheckErr(res);

   /* Realizing the Output Mix object in synchronous mode. */
   res = (*OutputMix)->Realize(OutputMix, SL_BOOLEAN_FALSE);
CheckErr(res);
   res = (*OutputMix)->GetInterface(OutputMix, SL_IID_VOLUME, (void
*)&volumeItf); CheckErr(res);

   /* Setup the data source structure */
   uri.locatorType         = SL_DATALOCATOR_URI;
   uri.pURI                 = "file:///music.wav";
   mime.formatType         = SL_DATAFORMAT_MIME;
   mime.pMimeType           = "audio/x-wav";
   mime.containerType      = SL_CONTAINERTYPE_WAV;

   audioSource.pLocator    = (void *)&uri;
   audioSource.pFormat     = (void *)&mime;

   /* Setup the data sink structure */
   locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
   locator_outputmix.outputMix     = OutputMix;
   audioSink.pLocator              = (void *)&locator_outputmix;
   audioSink.pFormat               = NULL;

   /* Set arrays required[] and iidArray[] for SEEK and EFFECTSEND
interface (PlayItf is implicit) */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_SEEK;
   required[1] = SL_BOOLEAN_TRUE;
   iidArray[1] = SL_IID_EFFECTSEND;

   /* Create the music player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, &player,
&audioSource, &audioSink, 2, iidArray, required); CheckErr(res);
```

```
   /* Realizing the player in synchronous mode. */
   res = (*player)->Realize(player, SL_BOOLEAN_FALSE); CheckErr(res);

   /* Get seek, play and effect send interfaces */
   res = (*player)->GetInterface(player, SL_IID_SEEK, (void *)&seekItf);
CheckErr(res);
   res = (*player)->GetInterface(player, SL_IID_PLAY, (void *)&playItf);
CheckErr(res);
   res = (*player)->GetInterface(player, SL_IID_EFFECTSEND, (void
*)&effectSendItf); CheckErr(res);


   /* Setup to receive position event callbacks */
   res = (*playItf)->RegisterCallback(playItf, PlayEventCallback, NULL);
   CheckErr(res);

   res = (*playItf)->SetPositionUpdatePeriod( playItf,
POSITION_UPDATE_PERIOD); CheckErr(res);
   res = (*playItf)->SetCallbackEventsMask( playItf,
SL_PLAYEVENT_HEADATNEWPOS); CheckErr(res);

   /* Before we start set volume to -3dB (-300mB) */
   res = (*volumeItf)->SetVolumeLevel(volumeItf, -300); CheckErr(res);

   /* Set Stereo Position to right channel but do not enable stereo
positioning */
   res = (*volumeItf)->SetStereoPosition(volumeItf, 1000); CheckErr(res);

   /* Get the Dynamic Interface Management interface for the Output Mix
object */
   res = (*OutputMix)->GetInterface(OutputMix,
SL_IID_DYNAMICINTERFACEMANAGEMENT, (void
*)&dynamicInterfaceManagementItf); CheckErr(res);
   /* Register DIM callback */
   res = (*dynamicInterfaceManagementItf)-
>RegisterCallback(dynamicInterfaceManagementItf, DIMCallback, NULL);
CheckErr(res);

   /* Play the music */
   res = (*playItf)->SetPlayState( playItf, SL_PLAYSTATE_PLAYING );
CheckErr(res);

   while ( (c = getchar()) != 'q' )
   {
      SLuint32 playState;
      SLPresetReverbItf PresetReverbItf;

      SLmillibel Min = SL_MILLIBEL_MIN;
      SLmillibel Max = 0;
      SLmillisecond PositionMSec = 0;
      SLboolean PresetReverbRealized = SL_BOOLEAN_FALSE;

      switch(c)
      {
         case '1':
            /* Play the music - only do so if it's not already playing
though */
            res = (*playItf)->GetPlayState( playItf, &playState );
CheckErr(res);
            if ( playState != SL_PLAYSTATE_PLAYING )
            {
               res = (*playItf)->SetPlayState(playItf,
SL_PLAYSTATE_PLAYING); CheckErr(res);
            }
            break;
         case '2':
            /* Pause the music - only do so if it is playing */
            res = (*playItf)->GetPlayState( playItf, &playState );
CheckErr(res);
            if ( playState == SL_PLAYSTATE_PLAYING )
            {
               res = (*playItf)->SetPlayState( playItf,
SL_PLAYSTATE_PAUSED); CheckErr(res);
            }
            break;
         case '3':
            /* Enable Stereo Positioning */
            res = (*volumeItf)->EnableStereoPosition(volumeItf,
SL_BOOLEAN_TRUE); CheckErr(res);
            break;
         case '4':
            /* Disable Stereo Positioning */
            res = (*volumeItf)->EnableStereoPosition(volumeItf,
SL_BOOLEAN_FALSE); CheckErr(res);
            break;
         case '5':
            /* Add some Reverb if not already present and there are
sufficient resources. */
            if (!PresetReverbRealized)
            {
               res = (*playItf)->GetPosition(playItf, &PositionMSec);
CheckErr(res);
               res = (*playItf)->GetPlayState( playItf, &playState );
CheckErr(res);
               /* We need to stop the music first though. */
               res = (*playItf)->SetPlayState(playItf,
SL_PLAYSTATE_STOPPED); CheckErr(res);

               /* Dynamically add the PresetReverb interface to the
Output Mix object */
               res = (*dynamicInterfaceManagementItf)-
>AddInterface(dynamicInterfaceManagementItf, SL_IID_PRESETREVERB,
SL_BOOLEAN_FALSE); CheckErr(res);
               /* Wait until asynchronous call terminates */
               sem_wait(&semDIM);
```

```
                if (asyncRes == SL_RESULT_SUCCESS)
                {
                   PresetReverbRealized = SL_BOOLEAN_TRUE;
                   /* Get PresetReverb interface */
                   res = (*OutputMix)->GetInterface(OutputMix,
SL_IID_PRESETREVERB, (void *)&PresetReverbItf);  CheckErr(res);
                   /* Setup PresetReverb for LARGE HALL */
                   res = (*PresetReverbItf)->SetPreset(PresetReverbItf,
SL_REVERBPRESET_LARGEHALL); CheckErr(res);
                   /* Enable the reverb effect and set the reverb level
for Audio Player at -3dB (-300mB) */
                   res = (*effectSendItf)->EnableEffectSend(effectSendItf,
PresetReverbItf, SL_BOOLEAN_TRUE, -300);
                }
                else
                {
                   /* Debug printing to be placed here */
                   exit(1);
                }

                /* Set head to continue from position where it was stopped
*/
                res = (*seekItf)->SetPosition(seekItf, PositionMSec,
SL_SEEKMODE_FAST); CheckErr(res);
                /* Go back to state before stopped */
                res = (*playItf)->SetPlayState(playItf, playState);
CheckErr(res);
             }
             break;
          case '6':
             /* Remove the Preset Reverb if present. */
             if (PresetReverbRealized)
             {
                res = (*playItf)->GetPosition(playItf, &PositionMSec);
CheckErr(res);
                res = (*playItf)->GetPlayState( playItf, &playState );
CheckErr(res);
                /* We need to stop the music first though. */
                res = (*playItf)->SetPlayState(playItf,
SL_PLAYSTATE_STOPPED); CheckErr(res);

                /* Disable the reverb effect for Audio Player */
                res = (*effectSendItf)->EnableEffectSend(effectSendItf,
PresetReverbItf, SL_BOOLEAN_FALSE, 0);

                /* Dynamically remove the PresetReverb interface from the
Output Mix object */
                res = (*dynamicInterfaceManagementItf)-
>RemoveInterface(dynamicInterfaceManagementItf, SL_IID_PRESETREVERB);
CheckErr(res);

                PresetReverbRealized = SL_BOOLEAN_FALSE;
```

```
               /* Set head to continue from position where it was stopped
*/
               res = (*seekItf)->SetPosition(seekItf, PositionMSec,
SL_SEEKMODE_FAST); CheckErr(res);
               /* Go back to state before stopped */
               res = (*playItf)->SetPlayState(playItf, playState);
CheckErr(res);
            }
            break;
         default:
            break;
      }
   }

   /* Stop the music */
   res = (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_STOPPED);
CheckErr(res);
   /* Destroy the player */
   (*player)->Destroy(player);
   /* Destroy Output Mix object */
   (*OutputMix)->Destroy(OutputMix);
}

int sl_main( void )
{
   SLresult    res;
   SLObjectItf sl;
   char c;

   SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };


   /* Simple test harness! */
   while ( (c = getchar()) != 'q' )
   {
      switch (c)
      {
         case '1':
            /* Create OpenSL ES */
            res = slCreateEngine( &sl, 3, EngineOption, 0, NULL, NULL);
CheckErr(res);
            /* Realizing the SL Engine in synchronous mode. */
            res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
            TestPlayMusicStereoPositionAndReverb(sl);
            /* Shutdown OpenSL ES */
            (*sl)->Destroy(sl);
            break;
         default:
            break;
        }
    }
    exit(0);
}
```

C.3 Phone Profile

This example shows how the API is used for the following case:

Listening to music on headset and incoming message alert requires MIDI to begin. Stop music playing on headset after storing final music play position.

Assume the application knows there is insufficient resources to have a MIDI player in existence at the same time as a Music player. So destroy the Music player and create a new MIDI player to generate message alert from file, playing audio using a different IODevice output (phone handsfree speaker).

When MIDI player has completed playing, destroy MIDI player and create music player to continue playing on headset from where it stopped.

C.3.1 Object Relationships

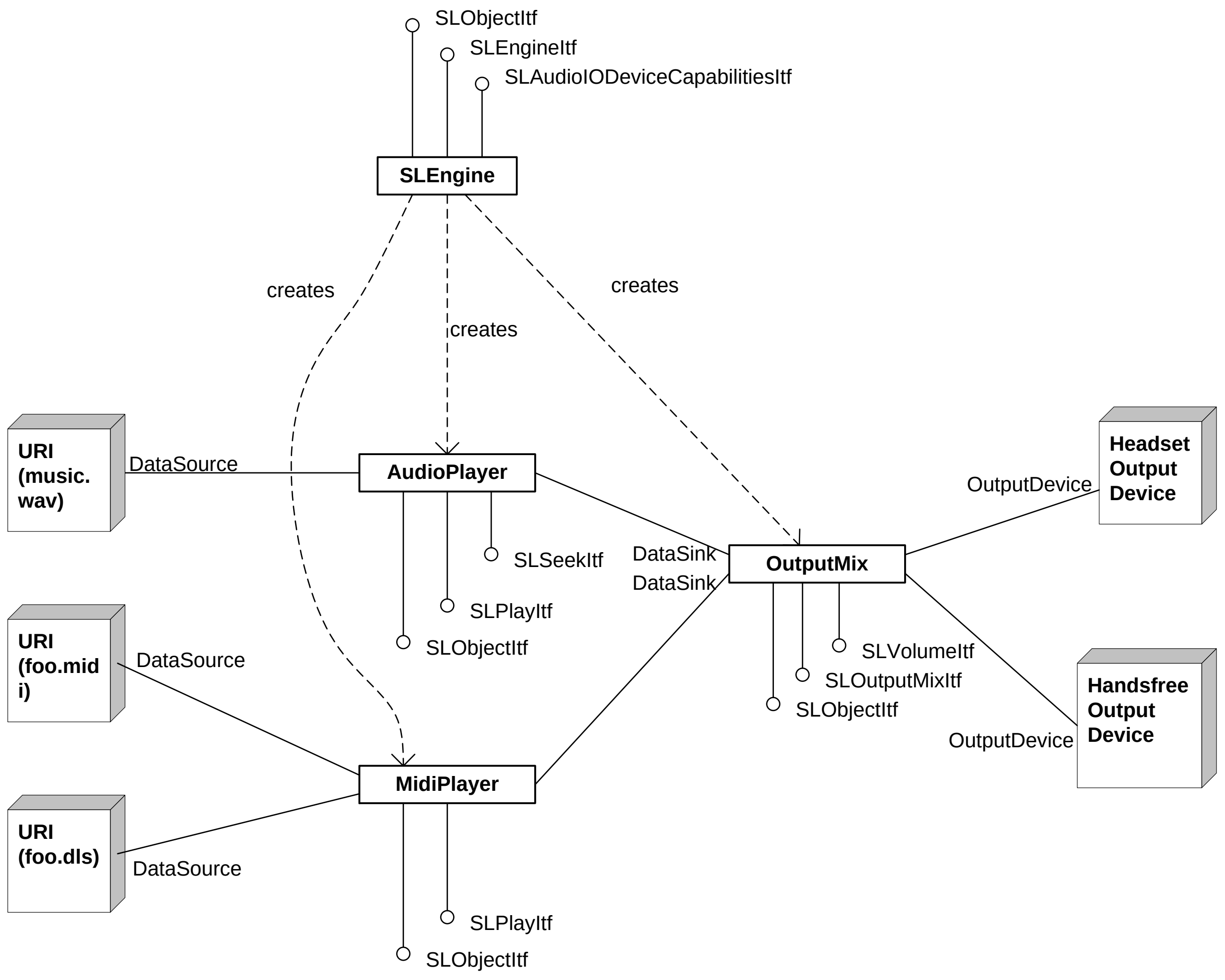


Figure 43: Object relationships(2)

C.3.2 Example Code

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

#define MAX_NUMBER_INTERFACES 3
#define MAX_NUMBER_OUTPUT_DEVICES 6

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      /* Debug printing to be placed here */
      exit(1);
   }
}

/* Listening to music in headphones and incoming message alert requires
    MIDI to begin.
    Stop music playing in headphones after storing final music play
    position.
    New MIDI player created to generate message alert from file to
    different IODevice output (ring speaker).
    When MIDI player completed playing then destroy Midi player and
    create music player to continue play from where it stopped. */

void TestPhone( SLObjectItf sl )
{
   SLEngineItf                      EngineItf;
   SLAudioIODeviceCapabilitiesItf AudioIODeviceCapabilitiesItf;
   SLAudioOutputDescriptor        AudioOutputDescriptor;

   SLuint32 OutputDeviceIDs[MAX_NUMBER_OUTPUT_DEVICES];
   SLint32   numOutputs = 0;
   SLboolean headset_available = SL_BOOLEAN_FALSE;
   SLboolean handsfree_speaker_available = SL_BOOLEAN_FALSE;
   SLuint32 headset_deviceID = 0;
   SLuint32 handsfree_speaker_deviceID = 0;
   SLuint32 deviceID = 0;

   SLresult                res;

   SLDataSource            audioSource;
   SLDataLocator_URI       uri;
   SLDataFormat_MIME       mime;

   SLDataSink              audioSink;
   SLDataLocator_OutputMix locator_outputmix;

   SLObjectItf             player;
   SLPlayItf               playItf;
   SLSeekItf               seekItf;

   SLObjectItf             OutputMix;
   SLOutputMixItf          outputMixItf;
   SLVolumeItf             volumeItf;

   int                     i;
   char                    c;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];

   SLmillisecond      DurationMsec = 0;
   /*Get the SL Engine Interface which is implicit*/
   res = (*sl)->GetInterface(sl, SL_IID_ENGINE, (void *)&EngineItf);
CheckErr(res);

   /* Initialize arrays required[] and iidArray[] */
   for (i=0;i<MAX_NUMBER_INTERFACES;i++)
   {
      required[i] = SL_BOOLEAN_FALSE;
      iidArray[i] = SL_IID_NULL;
   }

   /* Get the Audio IO DEVICE CAPABILITIES interface */
   res = (*sl)->GetInterface(sl, SL_IID_AUDIOIODEVICECAPABILITIES, (void
*)&AudioIODeviceCapabilitiesItf); CheckErr(res);
   numOutputs = MAX_NUMBER_OUTPUT_DEVICES;
   res = (*AudioIODeviceCapabilitiesItf)->GetAvailableAudioOutputs(
AudioIODeviceCapabilitiesItf, &numOutputs, OutputDeviceIDs);
CheckErr(res);

   /* Search for headset output and phone handsfree speaker device */
   for (i=0;i<numOutputs; i++)
   {
      res = (*AudioIODeviceCapabilitiesItf)-
>QueryAudioOutputCapabilities(AudioIODeviceCapabilitiesItf,
OutputDeviceIDs[i], &AudioOutputDescriptor); CheckErr(res);
      if((AudioOutputDescriptor.deviceConnection ==
SL_DEVCONNECTION_ATTACHED_WIRED)&&
          (AudioOutputDescriptor.deviceScope == SL_DEVSCOPE_USER)&&
          (AudioOutputDescriptor.deviceLocation ==
SL_DEVLOCATION_HEADSET))
      {
         headset_deviceID = OutputDeviceIDs[i];
         headset_available = SL_BOOLEAN_TRUE;
      }
      else if((AudioOutputDescriptor.deviceConnection ==
SL_DEVCONNECTION_INTEGRATED)&&
                  (AudioOutputDescriptor.deviceScope ==
SL_DEVSCOPE_ENVIRONMENT)&&
                  (AudioOutputDescriptor.deviceLocation ==
SL_DEVLOCATION_HANDSET))
      {
         handsfree_speaker_deviceID = OutputDeviceIDs[i];
         handsfree_speaker_available = SL_BOOLEAN_TRUE;
      }
   }


   /* Expect both headset output and phone handsfree speaker to be
available */
   if(!(headset_available && handsfree_speaker_available))
   {
      /* Debug printing to be placed here */
      exit(1);
   }

   /* Set arrays required[] and iidArray[] for VOLUME interface */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_VOLUME;

   /* Create Output Mix object to be used by player */
   res = (*EngineItf)->CreateOutputMix(EngineItf, &OutputMix, 1,
iidArray, required); CheckErr(res);

   /* Realizing the Output Mix object in synchronous mode. */
   res = (*OutputMix)->Realize(OutputMix, SL_BOOLEAN_FALSE);
CheckErr(res);
   res = (*OutputMix)->GetInterface(OutputMix, SL_IID_VOLUME, (void
*)&volumeItf); CheckErr(res);

   /* Get Output Mix interface */
   res = (*OutputMix)->GetInterface(OutputMix, SL_IID_OUTPUTMIX, (void
*)&outputMixItf);  CheckErr(res);
   /* Route output to headset */
   res = (*outputMixItf)->ReRoute(outputMixItf, 1, &headset_deviceID);
CheckErr(res);

   /* Setup the data source structure */
   uri.locatorType         = SL_DATALOCATOR_URI;
   uri.pURI                = "file:///music.wav";
   mime.formatType         = SL_DATAFORMAT_MIME;
   mime.pMimeType          = "audio/x-wav";
   mime.containerType      = SL_CONTAINERTYPE_WAV;

   audioSource.pLocator    = (void *)&uri;
   audioSource.pFormat     = (void *)&mime;

   /* Setup the data sink structure */
   locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
   locator_outputmix.outputMix     = OutputMix;
   audioSink.pLocator              = (void *)&locator_outputmix;
   audioSink.pFormat               = NULL;

   /* Set arrays required[] and iidArray[] for SEEK interface (PlayItf is
implicit) */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_SEEK;

   /* Create the music player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, &player,
&audioSource, &audioSink, 1, iidArray, required); CheckErr(res);

   /* Realizing the player in synchronous mode. */
   res = (*player)->Realize(player, SL_BOOLEAN_FALSE); CheckErr(res);
   /* Get seek and play interfaces */
   res = (*player)->GetInterface(player, SL_IID_SEEK, (void *)&seekItf);
CheckErr(res);
   res = (*player)->GetInterface(player, SL_IID_PLAY, (void *)&playItf);
CheckErr(res);

   /* Before we start set volume to -3dB (-300mB) */
   res = (*volumeItf)->SetVolumeLevel(volumeItf, -300); CheckErr(res);

   /* Get duration of content */
   res = (*playItf)->GetDuration(playItf, &DurationMsec); CheckErr(res);
   if (DurationMsec != SL_TIME_UNKNOWN)
   {
      /* Enable looping of entire file */
      res = (*seekItf)->SetLoop(seekItf, SL_BOOLEAN_TRUE, 0,
DurationMsec); CheckErr(res);
   }
   else
   {
      /* Debug printing to be placed here */
      exit(1);
   }

   /* Play the music */
   res = (*playItf)->SetPlayState( playItf, SL_PLAYSTATE_PLAYING );
CheckErr(res);

   while ( (c = getchar()) != 'q' )
   {
      SLuint32 playState;
      SLmillisecond PositionMSec = 0;
      SLboolean MidiPlayed = SL_BOOLEAN_FALSE;

      switch(c)
      {
         case '1':
            /* Begin playing Midi ringtone if not already playing after
stopping and destroying music player */
            if (!MidiPlayed)
            {
               SLObjectItf       Midi_player;
               SLPlayItf         Midi_playItf;
               SLDataSource      Midi_file;
               SLDataSource      Midi_bank;

               SLDataLocator_URI  Midi_fileLoc = { SL_DATALOCATOR_URI,
"file:///foo.mid"};
               SLDataFormat_MIME  Midi_fileFmt = { SL_DATAFORMAT_MIME,
"audio/x-midi", SL_CONTAINERTYPE_SMF };
               SLDataLocator_URI  Midi_bankLoc = { SL_DATALOCATOR_URI,
"file:///foo.dls"};
               SLDataFormat_MIME  Midi_bankFmt = { SL_DATAFORMAT_MIME,
"audio/dls", SL_CONTAINERTYPE_MOBILE_DLS };
```

```
               SLmillisecond     Midi_dur;
               SLmillisecond     Midi_pos;

               res = (*playItf)->GetPosition(playItf, &PositionMSec);
CheckErr(res);
               res = (*playItf)->GetPlayState( playItf, &playState );
CheckErr(res);
               /* Stop the music. */
               res = (*playItf)->SetPlayState(playItf,
SL_PLAYSTATE_STOPPED); CheckErr(res);
               /* Delete the music player */
               (*player)->Destroy(player);


               Midi_file.pFormat = (void*)&Midi_fileFmt;
               Midi_file.pLocator = (void*)&Midi_fileLoc;
               Midi_bank.pFormat = (void*)&Midi_bankFmt;
               Midi_bank.pLocator = (void*)&Midi_bankLoc;

               /* Create the Midi player */
               res = (*EngineItf)->CreateMidiPlayer(EngineItf,
&Midi_player, &Midi_file, &Midi_bank, &audioSink, NULL, NULL, 0, NULL,
NULL); CheckErr(res);

               /* Realizing the Midi player object in synchronous mode.
*/
               res = (*Midi_player)->Realize(Midi_player,
SL_BOOLEAN_FALSE); CheckErr(res);
               res = (*Midi_player)->GetInterface(Midi_player,
SL_IID_PLAY, (void *)&Midi_playItf); CheckErr(res);
               res = (*Midi_playItf)->GetDuration(Midi_playItf,
&Midi_dur); CheckErr(res);

               /* Route output to handsfree speaker */
               res = (*outputMixItf)->ReRoute(outputMixItf, 1,
&handsfree_speaker_deviceID); CheckErr(res);

               res = (*Midi_playItf)->SetPlayState(Midi_playItf,
SL_PLAYSTATE_PLAYING); CheckErr(res);
               do
               {
                  res = (*Midi_playItf)->GetPosition(Midi_playItf,
&Midi_pos); CheckErr(res);
               } while(Midi_pos < Midi_dur);

               MidiPlayed = SL_BOOLEAN_TRUE;

               /* Destroy Midi player */
               (*Midi_player)->Destroy(Midi_player);

               /* Create the music player */
               res = (*EngineItf)->CreateAudioPlayer(EngineItf, &player,
&audioSource, &audioSink, 1, iidArray, required); CheckErr(res);

               /* Realizing the player in synchronous mode. */
               res = (*player)->Realize(player, SL_BOOLEAN_FALSE);
CheckErr(res);

               /* Get seek and play interfaces *
               res = (*player)->GetInterface(player, SL_IID_SEEK, (void
*)&seekItf); CheckErr(res);
               res = (*player)->GetInterface(player, SL_IID_PLAY, (void
*)&playItf); CheckErr(res);

               /* Route output to headset *
               res = (*outputMixItf)->ReRoute(outputMixItf, 1,
&headset_deviceID); CheckErr(res);

               /* Set head to continue from position where it was stopped
*/
               res = (*seekItf)->SetPosition(seekItf, PositionMSec,
SL_SEEKMODE_FAST); CheckErr(res);
               /* Go back to state before stopped i.e. continue playing
music if not already at end of file */
               res = (*playItf)->SetPlayState(playItf, playState);
CheckErr(res);
            }
            break;

      }
   }

   /* Stop the music */
   res = (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_STOPPED);
CheckErr(res);
   /* Delete the player */
   (*player)->Destroy(player);
   /* Destroy Output Mix object */
   (*OutputMix)->Destroy(OutputMix);
}

int sl_main( void )
{
   SLresult    res;
   SLObjectItf sl;
   char c;

   SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };
   /* Simple test harness! */
   while ( (c = getchar()) != 'q' )
   {
      switch (c)
      {
         case '2':
            /* Create OpenSL ES */
            res = slCreateEngine( &sl, 3, EngineOption, 0, NULL, NULL);
CheckErr(res);
            /* Realizing the SL Engine in synchronous mode. */
            res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
            TestPhone(sl);
            /* Shutdown OpenSL ES */
            (*sl)->Destroy(sl);
            break;
         default:
            break;
      }
   }
   exit(0);
}
```

C.4 Game Profile

This example shows how the API is used in the following case:

There is a stationary 3D positioned Midi sound source, two PCM sound sources which are the sounds of a car engine noise and siren moving at a speed of 50kph from left to right. The Listener is stationary, looking forward.

C.4.1 Object Relationships

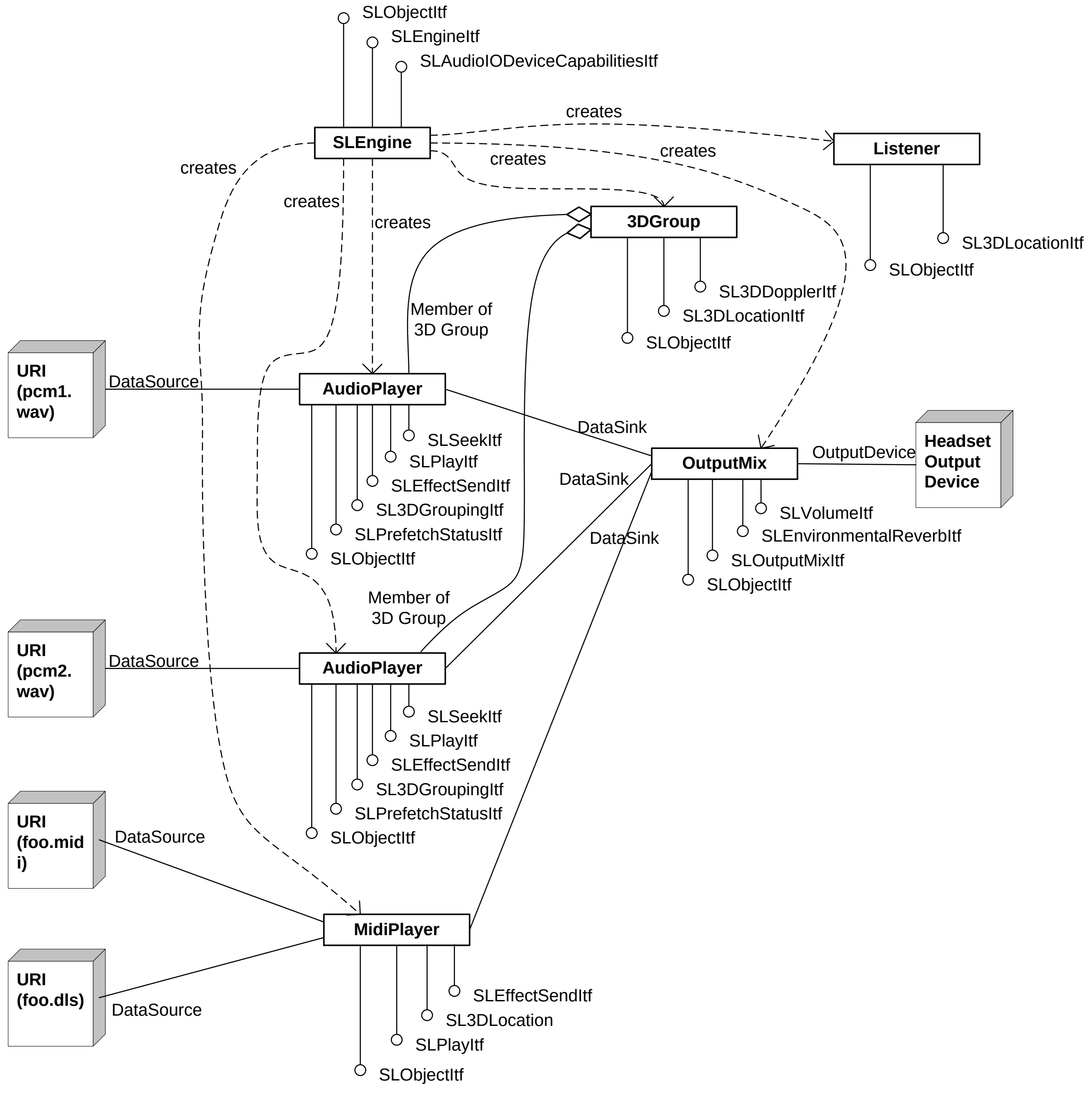


Figure 44: Object relationships(3)

C.4.2 Example Code

```
#include <stdio.h>
#include <stdlib.h>

#include <SLES/OpenSLES.h>

#define SLEEP(x)    /* Client system sleep function to sleep x
milliseconds would replace SLEEP macro */

#define MAX_NUMBER_INTERFACES 3
#define MAX_NUMBER_OUTPUT_DEVICES 6
#define CAR_SPEED_KPH 50  /* Speed of car is 50km/hour */
#define CAR_SPEED_MMPSEC CAR_SPEED_KPH*1000000/3600  /* Speed of car in
mm/second */

/* Checks for error. If any errors exit the application! */
void CheckErr( SLresult res )
{
   if ( res != SL_RESULT_SUCCESS )
   {
      /* Debug printing to be placed here */
      exit(1);
   }
}

/* Stationary 3D positioned Midi , 2 PCM sources car engine noise and
siren moving fast from left to right,
    Listener stationary looking forward
*/
void TestGame( SLObjectItf sl )
{
   SLEngineItf                      EngineItf;
   SLAudioIODeviceCapabilitiesItf  AudioIODeviceCapabilitiesItf;
   SLAudioOutputDescriptor         AudioOutputDescriptor;

   SLuint32 OutputDeviceIDs[MAX_NUMBER_OUTPUT_DEVICES];
   SLint32  numOutputs = 0;
   SLboolean headset_available = SL_BOOLEAN_FALSE;
   SLuint32 headset_deviceID = 0;
   SLuint32 deviceID = 0;

   SLresult                res;

   SLObjectItf             Midi_player;
   SLPlayItf               Midi_playItf;
   SLEffectSendItf         Midi_effectSendItf;
   SL3DLocationItf         Midi_3DLocationItf;

   SLDataSource            midSrc;
   SLDataSource            bnkSrc;
   SLDataLocator_URI  Midi_fileLoc = { SL_DATALOCATOR_URI,
"file:///foo.mid"};
   SLDataFormat_MIME  Midi_fileFmt = { SL_DATAFORMAT_MIME, "audio/x-
midi", SL_CONTAINERTYPE_SMF };
   SLDataLocator_URI  Midi_bankLoc = { SL_DATALOCATOR_URI,
"file:///foo.dls"};
   SLDataFormat_MIME  Midi_bankFmt = { SL_DATAFORMAT_MIME, "audio/dls",
SL_CONTAINERTYPE_MOBILE_DLS };

   SLObjectItf         Pcm1_player;
   SLObjectItf         Pcm2_player;
   SLPlayItf          Pcm1_playItf;
   SLPlayItf          Pcm2_playItf;
   SLSeekItf          Pcm1_seekItf;
   SLSeekItf          Pcm2_seekItf;
   SLPrefetchStatusItf      Pcm1_prefetchItf;
   SLPrefetchStatusItf      Pcm2_prefetchItf;
   SL3DGroupingItf    Pcm1_3DGroupingItf;
   SL3DGroupingItf    Pcm2_3DGroupingItf;
   SLEffectSendItf    Pcm1_effectSendItf;
   SLEffectSendItf    Pcm2_effectSendItf;

   SLDataLocator_URI   pcm1Loc = {SL_DATALOCATOR_URI,
"file:///pcm1.wav"};
   SLDataLocator_URI   pcm2Loc = {SL_DATALOCATOR_URI,
"file:///pcm2.wav"};
   SLDataFormat_MIME   pcmFormat = {SL_DATAFORMAT_MIME, "audio/x-wav",
SL_CONTAINERTYPE_WAV };
   SLDataSource        pcm1Src;
   SLDataSource        pcm2Src;

   SLObjectItf        Pcm_3DGroup;
   SL3DLocationItf    Pcm_3DLocationItf;
   SL3DDopplerItf     Pcm_3DDopplerItf;
   SLVec3D             Location = {-500000,5000,0};  /* 500 meters to the
left of origin, 5 meters in front of origin */
   SLVec3D             Midi_Location = {5000,-3000,3000};  /* 5 meters to
the right of origin, 3 meters behind origin, 3 meters above origin */
   SLVec3D             StartVelocity = {CAR_SPEED_MMPSEC,0,0};

   SLObjectItf         GameListener;
   SL3DLocationItf     Listener_3DLocationItf;

   SLVec3D             Listener_Front ={0,0,-1000};  /* Vector for having
listener look forward */
   SLVec3D             Listener_Above ={0,1000,0};   /* Vector for having
listener look forward */


   SLObjectItf         OutputMix;
   SLVolumeItf         volumeItf;
   SLOutputMixItf      outputMixItf;
   SLEnvironmentalReverbItf      EnvReverbItf;
   SLEnvironmentalReverbSettings ReverbSettings =
SL_I3DL2_ENVIRONMENT_PRESET_CITY;


   SLDataLocator_OutputMix locator_outputmix;
   SLDataSink              audioSink;

   int                 i;

   SLboolean required[MAX_NUMBER_INTERFACES];
   SLInterfaceID iidArray[MAX_NUMBER_INTERFACES];
   SLmillisecond      MidiDurationMsec = 0;
   SLmillisecond      PcmDurationMsec = 0;

   /* Get the SL Engine Interface which is implicit*/
   res = (*sl)->GetInterface(sl, SL_IID_ENGINE, (void *)&EngineItf);
CheckErr(res);

   /* Initialize arrays required[] and iidArray[] */
   for (i=0;i<MAX_NUMBER_INTERFACES;i++)
   {
      required[i] = SL_BOOLEAN_FALSE;
      iidArray[i] = SL_IID_NULL;
   }

   /* Get the Audio IO DEVICE CAPABILITIES interface */
   res = (*sl)->GetInterface(sl, SL_IID_AUDIOIODEVICECAPABILITIES, (void
*)&AudioIODeviceCapabilitiesItf); CheckErr(res);
   numOutputs = MAX_NUMBER_OUTPUT_DEVICES;
   res = (*AudioIODeviceCapabilitiesItf)->GetAvailableAudioOutputs(
AudioIODeviceCapabilitiesItf, &numOutputs, OutputDeviceIDs);
CheckErr(res);

   /* Search for headset output device */
   for (i=0;i<numOutputs; i++)
   {
      res = (*AudioIODeviceCapabilitiesItf)-
>QueryAudioOutputCapabilities(AudioIODeviceCapabilitiesItf,
OutputDeviceIDs[i], &AudioOutputDescriptor); CheckErr(res);
      if((AudioOutputDescriptor.deviceConnection ==
SL_DEVCONNECTION_ATTACHED_WIRED)&&
          (AudioOutputDescriptor.deviceScope == SL_DEVSCOPE_USER)&&
          (AudioOutputDescriptor.deviceLocation ==
SL_DEVLOCATION_HEADSET))
      {
         headset_deviceID = OutputDeviceIDs[i];
         headset_available = SL_BOOLEAN_TRUE;
         break;
      }
   }

   /* Expect headset output to be available */
   if(!headset_available)
   {
      /* Debug printing to be placed here */
      exit(1);
   }

   /* Set arrays required[] and iidArray[] for VOLUME and ENVIRONMENTAL
REVERB interface (OUTPUTMIX is implicit) */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_VOLUME;
   required[1] = SL_BOOLEAN_TRUE;
   iidArray[1] = SL_IID_ENVIRONMENTALREVERB;

   /* Create Output Mix object to be used by all players */
   res = (*EngineItf)->CreateOutputMix(EngineItf, &OutputMix, 2,
iidArray, required); CheckErr(res);

   /* Realizing the Output Mix object in synchronous mode. */
   res = (*OutputMix)->Realize(OutputMix, SL_BOOLEAN_FALSE);
CheckErr(res);

   res = (*OutputMix)->GetInterface(OutputMix, SL_IID_VOLUME, (void
*)&volumeItf); CheckErr(res);

   /* Get the environmental reverb interface */
   res = (*OutputMix)->GetInterface(OutputMix,
SL_IID_ENVIRONMENTALREVERB, (void *)&EnvReverbItf); CheckErr(res);

   /* Set reverb environment to city. */
   res = (*EnvReverbItf)->SetEnvironmentalReverbProperties(EnvReverbItf,
&ReverbSettings); CheckErr(res);

   /* Get Output Mix interface */
   res = (*OutputMix)->GetInterface(OutputMix, SL_IID_OUTPUTMIX, (void
*)&outputMixItf);  CheckErr(res);
   /* Route output to headset */
   res = (*outputMixItf)->ReRoute(outputMixItf, 1, &headset_deviceID);
CheckErr(res);

   /* Set up the MIDI data source */
   midSrc.pLocator = (void*)&Midi_fileLoc;
   midSrc.pFormat = (void*)&Midi_fileFmt;

   /* Set up the bank data source */
   bnkSrc.pLocator = (void*)&Midi_bankLoc;
   bnkSrc.pFormat = (void*)&Midi_bankFmt;

   /* Setup the data sink structure */
   locator_outputmix.locatorType   = SL_DATALOCATOR_OUTPUTMIX;
   locator_outputmix.outputMix     = OutputMix;
   audioSink.pLocator              = (void *)&locator_outputmix;
   audioSink.pFormat               = NULL;
   /* Create the Midi player */
   /* Set arrays required[] and iidArray[] for 3DLOCATION and EFFECTSEND
interfaces (PlayItf is implicit) */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_3DLOCATION;
   required[1] = SL_BOOLEAN_TRUE;
   iidArray[1] = SL_IID_EFFECTSEND;

   res = (*EngineItf)->CreateMidiPlayer(EngineItf, &Midi_player, &midSrc,
&bnkSrc, &audioSink, NULL, NULL, 2, iidArray, required); CheckErr(res);

   /* Realizing the Midi player object in synchronous mode. */
   res = (*Midi_player)->Realize(Midi_player, SL_BOOLEAN_FALSE);
CheckErr(res);

  /* Get playback, 3D location and effectsend interfaces
     for Midi player */
  res = (*Midi_player)->GetInterface(Midi_player, SL_IID_PLAY, (void
*)&Midi_playItf); CheckErr(res);
  res = (*Midi_player)->GetInterface(Midi_player, SL_IID_3DLOCATION,
(void *)&Midi_3DLocationItf); CheckErr(res);
  res = (*Midi_player)->GetInterface(Midi_player, SL_IID_EFFECTSEND,
(void *)&Midi_effectSendItf); CheckErr(res);

   /* Get duration of Midi content */
   res = (*Midi_playItf)->GetDuration(Midi_playItf, &MidiDurationMsec);
CheckErr(res);

   /* Set 3D location of Midi */
   res = (*Midi_3DLocationItf)->SetLocationCartesian(Midi_3DLocationItf,
&Midi_Location); CheckErr(res);

   /* Enable the reverb effect and set the reverb level for Midi Player
at -3dB (-300mB) */
   res = (*Midi_effectSendItf)->EnableEffectSend(Midi_effectSendItf,
EnvReverbItf, SL_BOOLEAN_TRUE, -300);

   /* Setup the data source structures for pcm1 and pcm2 */
   pcm1Src.pLocator = (void *)&pcm1Loc;
   pcm2Src.pLocator = (void *)&pcm2Loc;
   pcm1Src.pFormat = (void *)&pcmFormat;
   pcm2Src.pFormat = (void *)&pcmFormat;

   /* Set arrays required[] and iidArray[] for PREFETCH, SEEK, 3DGROUPING
and EFFECTSEND interfaces (PlayItf is implicit) */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_PREFETCHSTATUS;
   required[1] = SL_BOOLEAN_TRUE;
   iidArray[1] = SL_IID_3DGROUPING;
   required[2] = SL_BOOLEAN_TRUE;
   iidArray[2] = SL_IID_SEEK;
   required[3] = SL_BOOLEAN_TRUE;
   iidArray[3] = SL_IID_EFFECTSEND;
   /* Create the pcm1 player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, &Pcm1_player,
&pcm1Src, &audioSink, 4, iidArray, required); CheckErr(res);

   /* Realizing the pcm1 player in synchronous mode. */
   res = (*Pcm1_player)->Realize(Pcm1_player, SL_BOOLEAN_FALSE);
CheckErr(res);

   /* Get playback, prefetch, seek, 3D grouping and effect send
interfaces for Pcm1 player */
   res = (*Pcm1_player)->GetInterface(Pcm1_player, SL_IID_PLAY, (void
*)&Pcm1_playItf); CheckErr(res);
   res = (*Pcm1_player)->GetInterface(Pcm1_player, SL_IID_PREFETCHSTATUS,
(void *)&Pcm1_prefetchItf); CheckErr(res);
   res = (*Pcm1_player)->GetInterface(Pcm1_player, SL_IID_SEEK, (void
*)&Pcm1_seekItf); CheckErr(res);
   res = (*Pcm1_player)->GetInterface(Pcm1_player, SL_IID_3DGROUPING,
(void *)&Pcm1_3DGroupingItf); CheckErr(res);
   res = (*Pcm1_player)->GetInterface(Pcm1_player, SL_IID_EFFECTSEND,
(void *)&Pcm1_effectSendItf); CheckErr(res);

   /* Get duration of pcm1 content */
   res = (*Pcm1_playItf)->GetDuration(Pcm1_playItf, &PcmDurationMsec);
CheckErr(res);
   if (PcmDurationMsec != SL_TIME_UNKNOWN)
   {
      /* Enable looping of entire file */
      res = (*Pcm1_seekItf)->SetLoop(Pcm1_seekItf, SL_BOOLEAN_TRUE, 0,
PcmDurationMsec); CheckErr(res);
   }
   else
   {
      /* Debug printing to be placed here */
      exit(1);
   }

   /* Enable the reverb effect and set the reverb level for Pcm1 Player
at -3dB (-300mB) */
   res = (*Pcm1_effectSendItf)->EnableEffectSend(Pcm1_effectSendItf,
EnvReverbItf, SL_BOOLEAN_TRUE, -300);

   /* Create the pcm2 player */
   res = (*EngineItf)->CreateAudioPlayer(EngineItf, &Pcm2_player,
&pcm2Src, &audioSink, 4, iidArray, required); CheckErr(res);

   /* Realizing the pcm2 player in synchronous mode. */
   res = (*Pcm2_player)->Realize(Pcm2_player, SL_BOOLEAN_FALSE);
CheckErr(res);

   /* Get playback, prefetch, 3D grouping and effect send interfaces for
Pcm2 player */
```

```
   res = (*Pcm2_player)->GetInterface(Pcm2_player, SL_IID_PLAY, (void
*)&Pcm2_playItf); CheckErr(res);
   res = (*Pcm2_player)->GetInterface(Pcm2_player, SL_IID_PREFETCHSTATUS,
(void *)&Pcm2_prefetchItf); CheckErr(res);
   res = (*Pcm2_player)->GetInterface(Pcm2_player, SL_IID_SEEK, (void
*)&Pcm2_seekItf); CheckErr(res);
   res = (*Pcm2_player)->GetInterface(Pcm2_player, SL_IID_3DGROUPING,
(void *)&Pcm2_3DGroupingItf); CheckErr(res);
   res = (*Pcm2_player)->GetInterface(Pcm2_player, SL_IID_EFFECTSEND,
(void *)&Pcm2_effectSendItf); CheckErr(res);

   /* Get duration of pcm2 content */
   res = (*Pcm2_playItf)->GetDuration(Pcm2_playItf, &PcmDurationMsec);
CheckErr(res);
   if (PcmDurationMsec != SL_TIME_UNKNOWN)
   {
      /* Enable looping of entire file */
      res = (*Pcm2_seekItf)->SetLoop(Pcm2_seekItf, SL_BOOLEAN_TRUE, 0,
PcmDurationMsec); CheckErr(res);
   }
   else
   {
      /* Debug printing to be placed here */
      exit(1);
   }

   /* Enable the reverb effect and set the reverb level for Pcm2 Player
at -3dB (-300mB) */
  res = (*Pcm2_effectSendItf)->EnableEffectSend(Pcm2_effectSendItf,
EnvReverbItf, SL_BOOLEAN_TRUE, -300);

   /* Set arrays required[] and iidArray[] for 3DDoppler interface
(3DLocation is implicit) */
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_3DDOPPLER;

   /* Create 3DGroup to be used for pcm1 and pcm2 */
   res = (*EngineItf)->Create3DGroup(EngineItf, &Pcm_3DGroup, 1,
iidArray, required); CheckErr(res);

   /* Realizing the 3DGroup in synchronous mode. */
   res = (*Pcm_3DGroup)->Realize(Pcm_3DGroup, SL_BOOLEAN_FALSE);
CheckErr(res);
   res = (*Pcm_3DGroup)->GetInterface(Pcm_3DGroup, SL_IID_3DLOCATION,
(void *)&Pcm_3DLocationItf); CheckErr(res);
   res = (*Pcm_3DGroup)->GetInterface(Pcm_3DGroup, SL_IID_3DDOPPLER,
(void *)&Pcm_3DDopplerItf); CheckErr(res);

   /* Add pcm1 and pcm2 players to 3DGroup */

   res = (*Pcm1_3DGroupingItf)->Set3DGroup(Pcm1_3DGroupingItf,
Pcm_3DGroup); CheckErr(res);
```

```
   res = (*Pcm2_3DGroupingItf)->Set3DGroup(Pcm2_3DGroupingItf,
Pcm_3DGroup); CheckErr(res);

   /* Set arrays required[] and iidArray[] for 3DDoppler interface
(3DLocation is implicit) *
   required[0] = SL_BOOLEAN_TRUE;
   iidArray[0] = SL_IID_3DLOCATION;

   /* Create Listener */
   res = (*EngineItf)->CreateListener(EngineItf, &GameListener, 1,
iidArray, required); CheckErr(res);

   /* Realizing the Listener in synchronous mode. */
   res = (*GameListener)->Realize(GameListener, SL_BOOLEAN_FALSE);
CheckErr(res);
   res = (*GameListener)->GetInterface(GameListener, SL_IID_3DLOCATION,
(void *)&Listener_3DLocationItf); CheckErr(res);

   /* Set 3D orientation of Listener to look forward - even though this
is the default */
   res = (*Listener_3DLocationItf)-
>SetOrientationVectors(Listener_3DLocationItf, &Listener_Front,
&Listener_Above); CheckErr(res);

   /* Set location of 3Dgroup */
   res = (*Pcm_3DLocationItf)->SetLocationCartesian(Pcm_3DLocationItf,
&Location); CheckErr(res);

   /* Set velocity of 3Dgroup */
   res = (*Pcm_3DDopplerItf)->SetVelocityCartesian(Pcm_3DDopplerItf,
&StartVelocity); CheckErr(res);

   /* Place pcm1 and pcm2 players into Paused state to start prefetch */

   res = (*Pcm1_playItf)->SetPlayState(Pcm1_playItf,
SL_PLAYSTATE_PAUSED); CheckErr(res);
   res = (*Pcm2_playItf)->SetPlayState(Pcm2_playItf,
SL_PLAYSTATE_PAUSED); CheckErr(res);

   /* Wait until prefetch buffer is full for both pcm1 and pcm2 players
*/
   {
      SLpermille Pcm1FillLevel = 0;
      SLpermille Pcm2FillLevel = 0;
      while((Pcm1FillLevel != 1000)||(Pcm2FillLevel != 1000))
      {
         res = (*Pcm1_prefetchItf)->GetFillLevel(Pcm1_prefetchItf,
&Pcm1FillLevel); CheckErr(res);
         res = (*Pcm2_prefetchItf)->GetFillLevel(Pcm2_prefetchItf,
&Pcm2FillLevel); CheckErr(res);
      }
   }
   /* Start all 3 players */
   res = (*Midi_playItf)->SetPlayState(Midi_playItf,
SL_PLAYSTATE_PLAYING); CheckErr(res);
   res = (*Pcm1_playItf)->SetPlayState(Pcm1_playItf,
SL_PLAYSTATE_PLAYING); CheckErr(res);
   res = (*Pcm2_playItf)->SetPlayState(Pcm2_playItf,
SL_PLAYSTATE_PLAYING); CheckErr(res);

   /* Move the location of the 3Dgroup each second to maintain a speed of
50km/hour in direction of x-axis */
   {
      SLmillisecond Midi_pos;
      SLint32 XaxisLocation = Location.x;
      do
      {
         SLEEP(1000);    /* Delay 1000ms i.e. 1 second */

         XaxisLocation = Location.x + CAR_SPEED_MMPSEC;
         if(XaxisLocation <= 5000000)         /* Continue moving car
sounds until 5km away */
         {
            Location.x = XaxisLocation;
            res = (*Pcm_3DLocationItf)-
>SetLocationCartesian(Pcm_3DLocationItf, &Location); CheckErr(res);
         }
         res = (*Midi_playItf)->GetPosition(Midi_playItf, &Midi_pos);
CheckErr(res);
      } while(Midi_pos < MidiDurationMsec); /* continue until Midi file
finishes */
   }


   /* Stop the PCM players */
   res = (*Pcm1_playItf)->SetPlayState(Pcm1_playItf,
SL_PLAYSTATE_STOPPED); CheckErr(res);
   res = (*Pcm2_playItf)->SetPlayState(Pcm2_playItf,
SL_PLAYSTATE_STOPPED); CheckErr(res);


   /* Remove pcm1 and pcm2 players from 3DGroup */
   res = (*Pcm1_3DGroupingItf)->Set3DGroup(Pcm1_3DGroupingItf, NULL);
CheckErr(res);
   res = (*Pcm2_3DGroupingItf)->Set3DGroup(Pcm2_3DGroupingItf, NULL);
CheckErr(res);


   /* Destroy the objects */
   (*Pcm_3DGroup)->Destroy(Pcm_3DGroup);
   (*GameListener)->Destroy(GameListener);
   (*Pcm2_player)->Destroy(Pcm2_player);
   (*Pcm1_player)->Destroy(Pcm1_player);
   (*Midi_player)->Destroy(Midi_player);
   (*OutputMix)->Destroy(OutputMix);
```

```
}

int sl_main( void )
{

   SLresult    res;
   SLObjectItf sl;
   char c;

   SLEngineOption EngineOption[] = {
         (SLuint32) SL_ENGINEOPTION_THREADSAFE,
         (SLuint32) SL_BOOLEAN_TRUE,
         (SLuint32) SL_ENGINEOPTION_MAJORVERSION, (SLuint32) 1,
         (SLuint32) SL_ENGINEOPTION_MINORVERSION, (SLuint32) 1
   };


   /* Simple test harness! */
   while ( (c = getchar()) != 'q' )
   {
      switch (c)
      {
         case '3':
            /* Create OpenSL ES */
            res = slCreateEngine( &sl, 3, EngineOption, 0, NULL, NULL);
CheckErr(res);
            /* Realizing the SL Engine in synchronous mode. */
            res = (*sl)->Realize(sl, SL_BOOLEAN_FALSE); CheckErr(res);
            TestGame(sl);
            /* Shutdown OpenSL ES */
            (*sl)->Destroy(sl);
            break;
         default:
            break;
      }
   }

   exit(0);
}
```

Appendix D: Object-Interface Mapping

The following table describes the object-interface mapping per profile. It also shows mandated objects for each profile in its second row.

Object	Engine			Audio Player			MIDI player			Audio recorder			Listener			3D group			Output mix			Vibra			LED Array			Metadata Extractor		
Interface	P 🟦	M 🟦	G 🟦	P 🟦	M 🟦	G 🟦	P 🟦	M	G 🟦	P	M	G	P	M	G 🟦	P	M	G 🟦	P 🟦	M 🟦	G 🟦	P	M	G	P	M	G	P	M 🟦	G 🟦
SLObjectItf	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪
SLDynamicInterfaceManagementItf	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪
SLEngineItf	🟪	🟪	🟪																											
SLEngineCapabilitiesItf	🟪	🟪	🟪																											
SLThreadSyncItf	🟪	🟪	🟪																											
SLAudioIODeviceCapabilitiesItf	🟪	🟪	🟪																											
SLAudioDecoderCapabilitiesItf	🟦	🟦	🟦																											
SLAudioEncoderCapabilitiesItf										🟦	🟦	🟦																		
SLConfigExtensionsItf	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
SLLEDArrayItf																									🟪	🟪	🟪			
SLVibraItf																						🟪	🟪	🟪						
SLPlayItf				🟪	🟪	🟪	🟪	🟪	🟪																					
SLRecordItf										🟪	🟪	🟪																		
SLAudioEncoderItf										🟦	🟦	🟦																		
SLPrefetchStatusItf				🟦	🟦	🟦	🟦	🟩	🟦																					
SLSeekItf				A	🟦	🟦	A	🟩	🟦																					
SLPlaybackRateItf				🟩	🟩	🟩	🟩	🟩	🟩																					
SLRatePitchItf				🟩	🟩	🟦 1																								

Object	Engine			Audio Player			MIDI player			Audio recorder			Listener			3D group			Output mix			Vibra			LED Array			Metadata Extractor		
Interface	**P**	**M**	**G**	**P**	**M**	**G**	**P**	**M**	**G**	**P**	**M**	**G**	**P**	**M**	**G**	**P**	**M**	**G**	**P**	**M**	**G**	**P**	**M**	**G**	**P**	**M**	**G**	**P**	**M**	**G**
SLPitchItf				🟩	🟩	🟩	🟩	🟩	🟩																					
SLVolumeItf				B	🟦	🟦	B	🟩	🟦	🟩	🟩	🟩							B	🟦	🟦									
SLMuteSoloItf				🟩	🟩	🟦 1																								
SLBufferQueueItf						2			2	🟩	🟩	🟩																		
SLMIDIMessageItf							🟦	🟩	🟦																					
SLMIDITimeItf							C	🟩	🟦																					
SLMIDITempoItf							🟦	🟩	🟦																					
SLMIDIMuteSoloItf							🟩	🟩	🟦																					
SL3DCommitItf	🟩	🟩	🟦																											
SL3DGroupingItf				🟩	🟩	🟦	🟩	🟩	🟦																					
SL3DLocationItf				🟩	🟩	🟦	🟩	🟩	🟦				🟩	🟩	🟦	🟪	🟪	🟪												
SL3DSourceItf				🟩	🟩	🟦	🟩	🟩	🟦							🟩	🟩	🟦												
SL3DDopplerItf				🟩	🟩	🟦	🟩	🟩	🟦				🟩	🟩	🟦	🟩	🟩	🟦												
3DHintItf				🟩	🟩	🟩	🟩	🟩	🟩							🟩	🟩	🟩												
SL3DMacroscopicItf				🟩	🟩	🟩	🟩	🟩	🟩							🟩	🟩	🟩												
SLEffectSendItf				🟩	🟦	🟦	🟩	🟩	🟦																					
SLBassBoostItf				🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩							🟩	🟩	🟩									
SLEqualizerItf				🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩							🟩	🟦	🟦									
SLPresetReverbItf				🟩	🟩	🟩	🟩	🟩	🟩										🟩	🟦	🟩									
SLEnvironmentalReverbItf				🟩	🟩	🟩	🟩	🟩	🟩										🟩	🟩	🟦									

Object	Engine			Audio Player			MIDI player			Audio recorder			Listener			3D group			Output mix			Vibra			LED Array			Metadata Extractor		
Interface	P	M	G	P	M	G	P	M	G	P	M	G	P	M	G	P	M	G	P	M	G	P	M	G	P	M	G	P	M	G
SLVirtualizerItf				🟩	🟩	🟩	🟩	🟩	🟩										🟩	🟦	🟦									
SLMetadataExtractionItf				🟩	🟦	🟦	🟩	🟩	🟦																			🟪	🟪	🟪
SLMetadataMessageItf				🟩	🟦	🟦	🟩	🟩	🟦																			🟪	🟪	🟪
SLMetadataTraversalItf				🟩	🟦	🟦	🟩	🟩	🟦																			🟪	🟪	🟪
SLVisualizationItf				🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩							🟩	🟩	🟩									
SLOutputMixItf																			🟪	🟪	🟪									
SLDynamicSourceItf				🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩																🟪	🟪	🟪
SLDynamicSourceSinkChangeItf				🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩																🟪	🟪	🟪
SLDeviceVolumeItf	🟩	🟩	🟩																											

Legend	
P	Object mandated in Phone profile
M	Object mandated in Music profile
G	Object mandated in Game profile
P	Object optional in Phone profile
M	Object optional in Music profile
G	Object optional in Game profile
🟪	Implicit and mandated interface
🟦	Mandated (explicit) interface
1	Explicit interface mandated only in specifed circumstances, see comments.
A	Mandated (explicit) interface with some optional methods, see comments.
🟩	Applicable optional interfaces

Comments for explicit interfaces mandated only in specified circumstances:

1. This interface mandated for all players excluding those with Java Tone Sequences (JTS) data sources.
2. This interface is only mandated where the data source's locator is a buffer queue (`SLDataLocator_BufferQueue` or `SLDataLocator_MIDIBufferQueue`).

Comments for mandated interfaces with some optional methods:

A. Arbitrary loop points are not mandated in this profile, only end-to-end looping is mandated.

B. `SetStereoPosition()`, `GetStereoPosition()`, `EnableStereoPosition()` and `IsEnabledStereoPosition()` are not mandated in this profile.

C. `SetLoopPoints()` and `GetLoopPoints()` are not mandated in the profile.