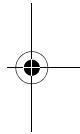
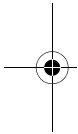


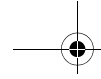
# Environmental Audio eXtensions™



**CREATIVE**®

WWW.SOUNDBLASTER.COM





**Copyright © 1998 by Creative Technology Ltd. All rights reserved.**

Version 1.0 (CLI), June 1998

Information in this document is subject to change without notice and does not represent a commitment on the part of Creative Technology Ltd. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the written permission of Creative Technology Ltd. The software described in this document is furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. It is against the law to copy the software on any other medium except as specifically allowed in the license agreement. The licensee may make one copy of the software for backup purposes.

#### **Trademarks**

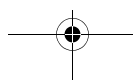
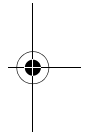
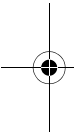
Creative, Sound Blaster, and the Creative logo are registered trademarks, and Environmetal Audio, E-mu Environmental Modeling, FourPointSurround, EMU10K1, Creative Multi Speaker Surround, EAX, Environmental Audio eXtensions, and the Sound Blaster Live! logo are trademarks of Creative Technology Ltd. in the United States and/or other countries.

E-mu, E-mu Systems, and SoundFont are registered trademarks of E-mu Systems, Inc.

Cambridge Soundworks is a registered trademark and PCWorks is a trademark of Cambridge Soundworks Inc., Newton, MA.

Microsoft, MS-DOS, Windows, DirectSound, DirectX, and DirectSound 3D are registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other brand and product names listed are trademarks or registered trademarks of their respective holders.

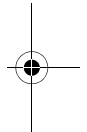
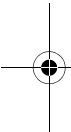


# Contents

<b>Introducing EAX .....</b>	<b>5</b>
DirectSound's Solutions .....	5
EAX's Solutions .....	6
3D Effects .....	6
Environments .....	7
An Open Standard .....	7
<b>How EAX Fits Into DirectSound .....</b>	<b>8</b>
DirectSound Property Sets .....	8
Primary and Secondary Sound Buffers .....	8
Sound Buffers .....	8
Primary Sound Buffer (The Listener) .....	9
Secondary Sound Buffers (Sound Sources) .....	9
Objects and Interfaces .....	10
EAX's Role In Direct Sound .....	10
Setting the Secondary-Buffer Property .....	10
Setting the Primary-Buffer Properties .....	11
<b>Creating Buffers and Interfaces .....</b>	<b>12</b>
<b>Setting EAX Properties .....</b>	<b>15</b>
Setting Properties Through a Property-Set Interface .....	15
Querying For EAX Support .....	15
Setting a Sound-Source Property .....	16
Setting a Listener Environment .....	16
Tweaking the Environment .....	17
Setting All the Listener Properties At Once .....	17
Setting a Reverb Preset .....	17
Getting a Property Value .....	19
Setting Properties Using the CReverb and	
CReverbBuffer Classes .....	19
CReverb Class .....	19
Constructor .....	19
Public Methods .....	20
CReverbBuffer .....	21



Constructor .....	21
Public Methods .....	21
Instantiating an EAX Class .....	22
Setting Individual Properties .....	22
Setting All Properties at One Time .....	22
Setting a Preset .....	23
<b>EAX Properties .....</b>	<b>24</b>
Sound-Source Property Set .....	24
Reverb Mix Property .....	24
Listener Property Set .....	25
Environment Property .....	25
Volume Property .....	27
Decay Time Property .....	27
Damping Property .....	28
<b>Creating an EAX Object .....</b>	<b>29</b>
<b>Future Plans for EAX .....</b>	<b>31</b>



# Environmental Audio eXtensions

Creative®'s Environmental Audio eXtensions (EAX™) are property sets of the Microsoft® DirectX™ game and multimedia programming environment for Windows® 95 . EAX adds 3D reverb capabilities to the DirectSound™ component of DirectX and provides an API (Application Programming Interface) that is an integral part of the DirectSound API. This document describes EAX, its place in DirectSound, its API, and how to use EAX in your own code.

## Introducing EAX

A primary goal of computer gaming is to produce a realistic 3D world for the player. Creating the aural component of that world has always lagged behind the visual component. It's not uncommon to find a computer displaying sophisticated 3D graphics with texture mapping, shading, multiple light sources, haze, and more while one or two tinny speakers spit out monaural sound.

### DirectSound's Solutions

Microsoft's DirectSound, the aural component of DirectX, provides a major first step for creating a realistic 3D aural world: it creates an easy-to-use programming environment for 3D aural modeling in C or C++. You can use DirectSound to create separate sound sources that move around realistically in the 3D aural world along with their corresponding objects in the 3D visual world—as the berserker warrior falls off the cliff, the player perceives his scream going with him.

DirectSound keeps the nitty gritty of audio hardware at arm's length from the programmer. The programmer uses DirectSound's relatively simple API to create sound sources, set their 3D positions and velocities (if moving), and take care of other decisions about the quality and placement of sounds in the aural world. DirectSound, through the audio driver installed in a computer, does the work of translating sound-source waveforms, positions, velocity, and more into a mix that ultimately comes out in realistic 3D form through the player's speakers or headphones.

Although DirectSound provides a number of sophisticated 3D aural effects such as Doppler effect, rolloff, interaural intensity and time differences, it lacks one very important effect: reverberation. Without reverb, a listener can tell where each sound source is located, but has no idea of the environment in which the sources are located. A sword clanked in a small padded cell should sound much different than the same sword clanked in a large cathedral, and it's reverb that tells the story. Without reverb, sound sources are naked and lack warmth—the aural equivalent of a visual world without shadows, haze, and independent light sources.

## EAX's Solutions

Creative's EAX property sets add reverb to DirectSound. As DirectSound property sets, they use the DirectSound API and the COM (Component Object Model) used throughout the DirectX environment. You can create an EAX interface for each sound source in the 3D aural world and set the reverb amount for individual sound sources. You can also control the overall quality of the reverb the listener hears, tweaking reverberation factors such as:

- Apparent size of the room surrounding the listener
- Amount and quality of the reverb's decay
- Volume of total reverberation.

These effects combine to add a visceral realism to DirectSound's 3D aural environment, an often subliminal context that can give an emotional depth to the 3D world of the player. All of this works even when the visual component of the 3D world is out of sight. Think, for example, of a single candle next to a pool of water in dark surroundings. When a drop of water hits the pool and you hear long and luscious reverb on the plink of the drop, your mind senses the vast cavern surrounding the pool even though you can't see it.

If you don't care for reverb in some circumstances, you can turn it off on a per-sound-source basis or turn it off altogether.

## 3D Effects

Because EAX is thoroughly integrated with DirectSound, it enhances the 3D aural world created by DirectSound. When you move a sound source in relation to the listener, EAX automatically adjusts the reverb for the sound source (increasing the ratio of reverb for a source moving away, for example) to make the reverb sound realistic for a moving sound source. None of DirectSound's 3D effects are lost in the mix; they're augmented with reverb calculated to enhance the feeling of three dimensions.



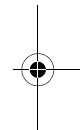
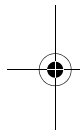
## Environments

EAX provides reverb environments that make it easy to simulate any one of a large variety of acoustic surroundings. Each EAX environment simulates a given set of acoustic surroundings such as an auditorium, a padded cell, an arena, a stone corridor, underwater, a city street, and so on. All you have to do is specify the environment you want. EAX takes care of the rest, supplying realistic 3D reverb for all the sound sources within the environment you choose. You can, if you wish, tweak the reverb quality of any environment to get the exact acoustic surroundings you want.

## An Open Standard

As DirectSound property sets, EAX is an open standard that takes advantage of any hardware-accelerated card (such as Sound Blaster® Live!) that provides the necessary reverb processing. When your application first asks for an EAX interface, DirectSound queries the card's audio driver to see if it supports property sets, then notifies the application so it can take advantage of EAX reverb if it's available.

As an open standard, EAX works not only with Creative's cards, but with any manufacturer's cards that care to take advantage of the EAX reverb property sets.



## How EAX Fits Into DirectSound

To use EAX well, it helps to understand its place in the DirectSound environment. Microsoft has extensive documentation explaining DirectSound, so we won't go into too many DirectSound details here, but will explore the main concepts you'll need to understand how EAX fits in.

### DirectSound Property Sets

If you're unfamiliar with property sets, Microsoft introduced them as an extension of DirectSound in release 5.0 of DirectX. Property sets are intended to provide access to advanced features of sound cards that aren't available through the DirectSound API. In other words, property sets make DirectSound extensible.

As the name suggests, a property set is a collection of properties, each with a setting or settings. An application checks first to see if the audio driver on the computer can support the property set. If it can, the application sets values for one, some, many, or all of the properties to control the effects provided by the property set.

A third-party hardware developer can create its own property set and then support the property set within the audio driver for an audio card. Each property set has a unique identifier, a *GUID* (Globally Unique Identifier), defined by the set creator, meant to be unique around the world for all time.

When an application wants to use a property set, it asks DirectSound for a property-set interface. The application may then query the interface, using the desired property set's GUID, to see if the sound card supports the property set. If the set is supported, the application can set a property by specifying the property set (by GUID), a property within that set (by number), and a value or array of values for that property. (It may also specify an instance parameter, but this is an uncommon occurrence.) The application can also get a property's current setting by specifying GUID and property number.

### Primary and Secondary Sound Buffers

Primary and secondary sound buffers are important parts of the DirectSound environment. They provide a mechanism for creating sound sources and a listener in a 3D aural world.

#### Sound Buffers

Sound buffers are used throughout DirectSound to contain a waveform table, a set of values that—converted to an analog signal—produce sound on an audio system. The application using DirectSound is responsible for providing the



waveform data for a sound buffer (but usually not the primary sound buffer) and then telling DirectSound how to play back that waveform data. An application can synthesize the waveform data for a sound buffer, insert a sound sample in the buffer, or stream audio to the buffer for playback.

### Primary Sound Buffer (the Listener)

DirectSound always has a single primary buffer—never more, never less—to feed waveform data directly to the audio system through a DAC (Digital-to-Audio Converter). Although the primary buffer provides only a single waveform table, it can support mono or stereo by interleaving samples for each channel within a single table.

Because the primary buffer is the direct waveform feed to the outside world and the player, it represents the *listener* in the 3D aural world, and is often referred to as such. DirectSound can (if set to work simulate a 3D environment) assign 3D settings to the primary buffer: a location, velocity, the listener's orientation (looking up, down, left, or right), and so on.

When an audio application runs, the primary buffer receives a mix of waveform data from other sound sources in the 3D aural world. DirectSound keeps track of the location of the other sound sources in relation to the listener and alters their output to simulate three dimensions. It may, for example, reduce a sound source's volume as it increases its distance from the listener, or add a Doppler effect if the source comes whooshing by the listener. An application rarely writes waveform data directly to the primary buffer, which is almost always under the direct control of DirectSound. If an application does take over writing to the primary buffer, it takes over the job of mixing sound sources.

DirectSound allows the application to be ignorant of primary buffer specifics: how and where the buffer is maintained, how the mix occurs. DirectSound handles the details of mixing sound sources, feeding the mix to the primary buffer, and working with the audio card to send the primary buffer contents to the outside world.

### Secondary Sound Buffers (Sound Sources)

DirectSound supports as many secondary sound buffers as the host system is able to accommodate in computer RAM or sound-card RAM. Each of these buffers represents a *sound source*. Each holds a waveform table created by the application for playback and plays back the waveform table as directed by the application. When DirectSound is set to simulate a 3D environment, the application can assign a secondary buffer's position in the 3D aural world as well as its velocity (if it's in motion), its sound cones (for directional sound projection), its minimum-maximum distances from the listener, and so on.

## Objects and Interfaces

Because DirectSound is object oriented, it creates each sound buffer—the primary and any secondary buffers—as objects. To provide application control for each sound object, DirectSound creates a standard interface (also an object) tied to each sound object. An application controls each sound buffer through the buffer's interface by calling methods on the interface. An application can, for example, call the method **SetVolume** on a buffer's standard interface to set the buffer's playback volume.

DirectSound gets a little more complicated when you use it to create a 3D aural world: it requires two interfaces for each buffer. The first interface is the standard buffer interface that sets non-3D buffer characteristics such as playback volume or frequency. The second interface is a 3D buffer interface that sets 3D buffer characteristics such as location or velocity.

When you want to use a property set with a 3D buffer, you work with three interfaces: the standard buffer interface for non-3D characteristics, the 3D buffer interface for 3D characteristics, and a property-set interface to set special buffer properties. The property-set interface allows the application to first query to see if a particular property set exists, and then—if it's there—to set property values within the set and to read current property values.

## EAX's Role In Direct Sound

EAX is comprised of two different property sets. The first, called the *sound-source property set*, applies directly to individual secondary buffers (sound sources). It has a single property. The second set, called the *listener property set*, applies only to the primary buffer (the listener). It has four properties.

### Setting the Secondary-Buffer Property

The single property in the sound-source property set (called *Reverb Mix*, described in detail later in this document) allows the application to set the ratio of the reverb to the sound source. Most applications don't set this property for a sound buffer because if it's not set, EAX maintains the reverb/source ratio automatically. To use the property with a secondary buffer, an application requests a property-set interface for the buffer. When it gets the interface, it calls the **Set** method on the interface and specifies the EAX sound-source GUID, specifies the Reverb Mix property by number, and provides a pointer to the ratio value it wants to set. That ratio value applies only to this secondary buffer and not to any other buffers.

## Setting the Primary-Buffer Properties

The four properties in the listener property set all work only on the primary buffer. Because of the way DirectSound works, you can only use this set in the property-set interface of a secondary buffer—not in the property-set interface of the primary buffer. You can use the property-set interface of *any* secondary buffer, but you must remember when using the four listener properties that they affect the primary buffer as it's seen by all other secondary buffers. In other words, if you set a primary-buffer property in one secondary buffer interface, that primary property is set to the same value in all other secondary buffer interfaces.

The properties in the listener property set control the quality of the reverb as perceived by the listener; one sets a reverb environment, the other three adjust that environment. Any one of these settings applies to all the mixed sound sources—but is scaled for each individual source depending on its position or its individual Reverb Mix setting. EAX adds reverb only after DirectSound has added all of its 3D effects, so the sound sources retain their original 3D quality and have it enhanced by 3D reverb effects.

To set primary-buffer properties, an application requests a property-set interface for a secondary buffer—or it can use an existing property-set interface if it received one earlier while setting secondary-buffer properties. The application calls the **Set** method on the interface and passes to it the listener property-set GUID, the property ID, and a pointer to the value it wants to set. EAX applies the new property setting to the primary buffer regardless of where the property was set.

## Creating Buffers and Interfaces

To use EAX in an application, you must first start DirectSound. You then create appropriate sound buffers and create three interfaces for each buffer:

- Standard interface to control fundamental buffer behavior (playback volume, frequency, and so on)
- 3D interface to control 3D behavior (position, velocity, and so on)
- Property-set interface to set EAX properties for the buffer

To do so, follow these steps, illustrated with sample code:

1. Instantiate a **DirectSound** object to start DirectSound, then use the **IDirectSound** interface to control the DirectSound object:

```
LPDIRECTSOUND pDirectSoundObj;
DirectSoundCreate(NULL, &pDirectSoundObj, NULL);
pDirectSoundObj->SetCooperativeLevel(hWnd, DSSCL_EXCLUSIVE);
```

Creating the DirectSound object gives the application a pointer to the interface for that object. You can use the interface as shown in line 3 of the sample above to set DirectSound's status: its cooperative level, number of speakers, and so forth.

2. Instantiate a new 3D primary buffer :

```
LPDIRECTSOUNDBUFFER pPrimaryBuf;
DSBUFFERDESC desc;
...
desc.dwFlags = DSBCAPS_PRIMARYBUFFER | DSBCAPS_CTRL3D;
pDirectSoundObj->CreateSoundBuffer(&desc, &pPrimaryBuf, NULL);
```

The flags **DSBCAPS\_PRIMARYBUFFER** and **DSBCAPS\_CTRL3D** specify a 3D primary buffer.

Creating the primary buffer gives the application a pointer to the buffer's standard interface. Note that the new buffer you create replaces the default primary buffer created when you instantiated the DirectSound object.

3. Get a 3D interface (called **IDirectSound3DListener**) for the new primary buffer:

```
LPDIRECTSOUND3DLISTENER pListener;
pPrimaryBuf->QueryInterface(IID_IDirectSound3DListener
(void**)&pListener);
```

You can use this interface to control listener-specific 3D aspects of the primary buffer behavior—listener orientation, position, velocity, and so forth.

4. Instantiate a secondary buffer for each sound source in the 3D aural world:

```
LPDIRECTSOUNDBUFFER pSecondaryBuf[n];
DSBUFFERDESC desc;
...
desc.dwFlags = DSBCAPS_CTRL3D | ...
pDirectSoundObj->CreateSoundBuffer(&desc, &pSecondaryBuf[i], NULL);
```

The flag **DSBCAPS\_CTRL3D** makes the buffer a 3D buffer.

Each secondary buffer creation gives the application a pointer to the buffer's standard interface.

5. Get a 3D-secondary-buffer interface (called **IDirectSound3DBuffer**) for each new secondary buffer:

```
LPDIRECTSOUND3DBUFFER p3Dbuf[n];
...
pSecondaryBuf[i]->QueryInterface(IID_IDirectSound3DBuffer,
    (void **)&p3Dbuf[i]);
```

Each of these interfaces controls 3D aspects of a secondary buffer's behavior.

6. Establish a property-set interface for the primary buffer (the listener). Because a DirectSound quirk doesn't allow an application to use a property set directly with the primary buffer interface, your application must use one of the secondary buffer's property-set interfaces to set EAX listener properties. You can get a pointer to an interface from DirectSound:

```
LPKSPROPERTYSET pReverb;
p3Dbuf[0]->QueryInterface(IID_IKsPropertySet, (void **)&pReverb);
```

You get a property-set interface for a buffer by calling **QueryInterface** on the buffer's 3D interface. It writes a pointer to the property-set interface. (This example uses the first secondary buffer you created, buffer 0. You could, in fact, use any secondary buffer.)

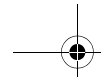
You can use the property-set interface to set EAX listener properties by specifying the EAX GUID and appropriate listener property numbers as described in "Setting EAX Properties" on page 15.

Note that if you ask for a property-set interface and the audio driver doesn't support property sets, **QueryInterface** will fail.

7. If you want to directly control the amount of a single sound source's reverb, get a property-set interface for the secondary buffer of each sound source you want to control:

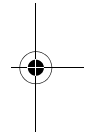
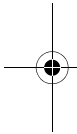
```
LPKSPROPERTYSET pReverbBuffer[n];
...
p3Dbuf[i]->QueryInterface(IID_IKsPropertySet, (void **)&pReverbBuffer[i]);
```

This example gets a property-set interface for each secondary buffer.



You can use each secondary buffer's property-set interface to set EAX properties by specifying the EAX GUID and a property number as described in "Setting EAX Properties" on page 15.

Note that most programs don't directly control the amount of each sound source's reverb; they leave control to EAX, which automatically sets reverb according to the sound source's distance from the listener. If your program doesn't need to control each sound source's reverb directly, then it can skip this step completely.



## Setting EAX Properties

Once you've created primary and secondary buffers and established three interfaces (standard, 3D, and property-set) for each buffer you want to control, you can set EAX properties in either of two ways:

- You can set properties directly through DirectSound's property-set interface
- You can set properties using instances of two C++ classes—**CReverb** and **CReverbBuffer**—that are supplied with EAX as header and source files for each class.

### Setting Properties Through a Property-Set Interface

A property-set interface offers three methods: **QuerySupport**, **Set**, and **Get**. These methods all require a GUID that identifies a property set and the ID number of a property in that set. When you use these methods with EAX, it's important to remember that EAX is really two property sets:

- The **EAX sound-source property set**, which applies to a sound source (a secondary buffer) and contains a single property. Its GUID is the constant **DSPROPSETID\_EAXBUFFER\_ReverbProperties**, defined in the header file EAX.H.
- The **EAX listener property set**, which applies to the listener (the primary buffer) and contains four properties. Its GUID is the constant **DSPROPSETID\_EAX\_ReverbProperties**, defined in the header file EAX.H.

### Querying for EAX Support

Before your application tries to work with EAX properties, you may want it to check to be sure that the audio driver supports the two EAX property sets. To do so, you call **QuerySupport** on the property-set interface of any buffer.

**QuerySupport** can query to see if either EAX property set exists or it can query to see if an individual EAX property is supported. In both cases, **QuerySupport** requires the application to pass it the GUID of the property set being queried along with a property number. If you want to query for the listener property set, pass the listener property set GUID along with the constant **DSPROPERTY\_EAX\_ALL** as the property number. If you want to query for the sound-source property set, pass the sound-source property set GUID along with the constant **DSPROPERTY\_EAXBUFFER\_ALL** as the property number.

The following sample code queries the EAX interface pointed to by **pReverb**. It asks if it supports the EAX listener property set, then checks to see if the

value returned includes flags that show support for getting and setting all properties within the set:

```
ULONG support=0;
if (FAILED(pReverb->QuerySupport(DSPROPSETID_EAX_ReverbProperties,
    DSPROPERTY_EAX_ALL, &support)))
    AfxMessageBox("EAX not supported");
if ((support & (KSPROPERTY_SUPPORT_GET|KSPROPERTY_SUPPORT_SET)) !=
    (KSPROPERTY_SUPPORT_GET|KSPROPERTY_SUPPORT_SET))
    AfxMessageBox("EAX not supported");
```

### Setting a Sound-Source Property

The single property in the EAX sound-source property set determines the reverb/source ratio for that sound source (as described in “Sound-Source Property Set” on page 24). To set it, you use the property-set interface of the buffer you want to affect. You then call **Set** on the interface:

```
float reverbMix=1.0F;
pReverbBuffer[i]
->Set(DSPROPSETID_EAXBUFFER_ReverbProperties,
    DSPROPERTY_EAXBUFFER_REVERBMIX, NULL, 0, &reverbMix, sizeof(float));
```

The first argument you pass is the GUID of the EAX sound-source property set. The second argument is a constant (defined in EAX.H) that identifies the Reverb Mix property (the only property in the set). The third and fourth arguments define a parameter, which isn’t used by EAX, so they receive NULL and 0. The fifth and sixth arguments pass the property’s new value. The fifth is a pointer to the value, the sixth gives the size of the value.

### Setting a Listener Environment

The most important of the four properties in the EAX listener property set is Environment. This property (described in “Listener Property Set” on page 25) defines the size and acoustic quality of the apparent room in which the listener is located. To set Environment, use the property-set interface of any secondary buffer. (The set affects the primary buffer’s operation, not the secondary buffer’s operation.) You call **Set** on the interface:

```
unsigned long envId=EAX_ENVIRONMENT_AUDITORIUM;
pReverb->Set(DSPROPSETID_EAX_ReverbProperties,
    DSPROPERTY_EAX_ENVIRONMENT,
    NULL,0,&envId,sizeof(unsigned long));
```

The first argument is the GUID of the EAX listener property set; the second is the ID of the property you want to set (a constant defined in EAX.H), in this case Environment. The remaining arguments are set to ignore parameter and point to a property value, just as in the previous example.

When you set Environment, EAX automatically sets values for the other three listener properties, which are defined internally as part of the environment.



## Tweaking the Environment

Once you set an environment, you can tweak the environment by setting the other three listener properties: Volume, Decay Time, Damping. Set them just as you do Environment. This example sets Volume:

```
float volume=0.5F;
pReverb->Set(DSPROPSETID_EAX_ReverbProperties, DSPROPERTY_EAX_VOLUME,
  NULL,0,&volume,sizeof(float));
```

## Setting All the Listener Properties at Once

To set all the listener properties at once, you call **Set** on a property-set interface just as you did in the last example, but the ID you use to specify a property—**DSPROPERTY\_EAX\_ALL**—specifies *all* listener properties in this case. And the value you pass to the **Set** method is an array of four values instead of a single value:

```
EAX_REVERBPROPERTIES props={EAX_ENVIRONMENT_AUDITORIUM, 0.5F,4.0F,0.5F};
pReverb->Set(DSPROPSETID_EAX_ReverbProperties, DSPROPERTY_EAX_ALL, NULL,
  0, &props, sizeof(EAX_REVERBPROPERTIES));
```

The values in the array apply to the listener properties in this order:

- **Environment** (DSPROPERTY\_EAX\_ENVIRONMENT)
- **Volume** (DSPROPERTY\_EAX\_VOLUME)
- **Decay Time** (DSPROPERTY\_EAX\_DECAYTIME)
- **Damping** (DSPROPERTY\_EAX\_DAMPING)

You can read more about these individual properties in “Listener Property Set” on page 25.

## Setting a Reverb Preset

The EAX.H file contains a set of defined presets. Each preset is an array of four listener property values in the same order used to set all properties at once (as just described). You may, if you like, define your own presets as well.

To use a preset, call **Set** on a property-set interface using the property **DSPROPERTY\_EAX\_ALL** just as in the previous example, but pass a preset constant in place of the array. Here’s an example using the custom preset **EAX\_PRESET\_AUDITORIUM**:

```
EAX_REVERBPROPERTIES preset={EAX_PRESET_AUDITORIUM};
pReverb->Set(DSPROPSETID_EAX_ReverbProperties, DSPROPERTY_EAX_ALL,
  NULL, 0,
  &preset, sizeof(EAX_REVERBPROPERTIES));
```

The following table lists by constant ID the presets available in the audio driver. Each preset is followed by the four listener property values that define each of the presets.

Preset	Base Environment	Volume	Decay	
			Time	Damping
EAX_PRESET_GENERIC	EAX_ENVIRONMENT_GENERIC	0.5F	1.493F	0.5F
EAX_PRESET_PADDEDCELL	EAX_ENVIRONMENT_PADDEDCELL	0.25F	0.1F	0.0F
EAX_PRESET_ROOM	EAX_ENVIRONMENT_ROOM	0.417F	0.4F	0.666F
EAX_PRESET_BATHROOM	EAX_ENVIRONMENT_BATHROOM	0.653F	1.499F	0.166F
EAX_PRESET_LIVINGROOM	EAX_ENVIRONMENT_LIVINGROOM	0.208F	0.478F	0.0F
EAX_PRESET_STONEROOM	EAX_ENVIRONMENT_STONEROOM	0.5F	2.309F	0.888F
EAX_PRESET_AUDITORIUM	EAX_ENVIRONMENT_AUDITORIUM	0.403F	4.279F	0.5F
EAX_PRESET_CONCERTHALL	EAX_ENVIRONMENT_CONCERTHALL	0.5F	3.961F	0.5F
EAX_PRESET_CAVE	EAX_ENVIRONMENT_CAVE	0.5F	2.886F	1.304F
EAX_PRESET_ARENA	EAX_ENVIRONMENT_ARENA	0.361F	7.284F	0.332F
EAX_PRESET_HANGAR	EAX_ENVIRONMENT_HANGAR	0.5F	10.0F	0.3F
EAX_PRESET_CARPETEDHALLWAY	EAX_ENVIRONMENT_CARPETEDHALLWAY	0.153F	0.259F	2.0F
EAX_PRESET_HALLWAY	EAX_ENVIRONMENT_HALLWAY	0.361F	1.493F	0.0F
EAX_PRESET_STONECORRIDOR	EAX_ENVIRONMENT_STONECORRIDOR	0.444F	2.697F	0.638F
EAX_PRESET_ALLEY	EAX_ENVIRONMENT_ALLEY	0.25F	1.752F	0.776F
EAX_PRESET_FOREST	EAX_ENVIRONMENT_FOREST	0.111F	3.145F	0.472F
EAX_PRESET_CITY	EAX_ENVIRONMENT_CITY	0.111F	2.767F	0.224F
EAX_PRESET_MOUNTAINS	EAX_ENVIRONMENT_MOUNTAINS	0.194F	7.841F	0.472F
EAX_PRESET_QUARRY	EAX_ENVIRONMENT_QUARRY	1.0F	1.499F	0.5F
EAX_PRESET_PLAIN	EAX_ENVIRONMENT_PLAIN	0.097F	2.767F	0.224F
EAX_PRESET_PARKINGLOT	EAX_ENVIRONMENT_PARKINGLOT	0.208F	1.652F	1.5F
EAX_PRESET_SEWERPIPE	EAX_ENVIRONMENT_SEWERPIPE	0.652F	2.886F	0.25F
EAX_PRESET_UNDERWATER	EAX_ENVIRONMENT_UNDERWATER	1.0F	1.499F	0.0F
EAX_PRESET_DRUGGED	EAX_ENVIRONMENT_DRUGGED	0.875F	8.392F	1.388F
EAX_PRESET_DIZZY	EAX_ENVIRONMENT_DIZZY	0.139F	17.234F	0.666F
EAX_PRESET_PSYCHOTIC	EAX_ENVIRONMENT_PSYCHOTIC	0.486F	7.563F	0.806F

Note that these presets are defined to set the default Volume, Decay Time, and Damping values that are associated with each Environment setting. For example, using the preset **EAX\_PRESET\_CONCERTHALL** is the same as setting Environment to **EAX\_ENVIRONMENT\_CONCERTHALL**. You can use the preset definitions to see what default values are set by each environment. You can also use the presets as starting points for your own preset definitions.

### Getting a Property Value

You can, at any time, check to see the current value of any EAX property. To do so, you call **Get** on a secondary buffer's property-set interface and identify the property you want by GUID and property ID. For example:

```
float volume;
ULONG volsize;
pReverb->Get(DSPROPSETID_EAX_ReverbProperties,
             DSPROPERTY_EAX_VOLUME, NULL,
             0, &volume, sizeof(float), &volsize);
```

This example specifies the Volume property in the listener property set. **Get** returns the value of Volume in the variable **volume** and writes the number of bytes it wrote into **volume** into the variable **volsize**.

### Setting Properties Using the CReverb and CReverbBuffer Classes

If you're programming in C++, you can use two C++ classes included with the EAX SDK: **CReverb** and **CReverbBuffer**. These two classes create EAX interfaces that make it simple to set EAX properties.

#### CReverb Class

**CReverb** defines an object that is an EAX interface for the primary buffer—the listener. It works through the 3D interface of a secondary buffer.

##### Constructor

```
CReverb(LPDIRECTSOUND3DBUFFER p3DBuf)
```

The constructor takes a pointer to the 3D interface of a 3D secondary buffer. Remember that this can be any secondary buffer; the settings that this class works with affect only the primary buffer, not the secondary buffer. The object works *through* the secondary buffer but not *on* it.

### *Public Methods*

**int PropertySetOk()**

This method checks to see if the audio driver supports the EAX listener property set. When called, it returns a TRUE if EAX is supported, a FALSE if not.

**void SetEnvironment(unsigned long envId)**

This method sets the Environment listener property. It accepts an unsigned long integer (typically an environment constant from the EAX.H file) that specifies the desired reverb environment. Remember that setting an Environment value by itself automatically sets the Volume, Decay Time, and Damping values to default values for the specified environment.

**void SetVolume(float volume)**

This method sets the Volume listener property. It accepts a floating point value from 0.0 to 1.0. (See “Volume Property” on page 27 for details on what this value sets.)

**void SetDecayTime(float time)**

This method sets the Decay Time listener property. It accepts a floating point value from 0.1 to 20.0. (See “Decay Time Property” on page 27 for details on what this value sets.)

**void SetDamping(float damping)**

This method sets the Damping listener property. It accepts a floating point value from 0.0 to 2.0. (See “Damping Property” on page 28 for details on what this value sets.)

**Void SetAll(EAX\_REVERBPROPERTIES \*pProperties)**

This method sets all four listener properties at one time. It accepts a pointer to a four-value array of property values. The order of the values must be Environment, Volume, Decay Time, and Damping.

**void SetPreset(unsigned long envId, float volume, float time, float damping)**

This method sets a preset, an array of all four property values defined in a header file. Although it looks here as if it accepts four different property values, in coding the method accepts a #defined preset constant that expands to the four values before compiling.

**unsigned long GetEnvironment()**

This method returns the current Environment property value.

**float GetVolume()**

This method returns the current Volume property value.

**float GetDecayTime()**

This method returns the current Decay Time property value.

```
float GetDamping()
```

This method returns the current Damping property value.

```
void GetAll(EAX_REVERBPROPERTIES *pProperties)
```

This method returns a pointer to all a four value array of all the current listener property values in the order Environment, Volume, Decay Time, Damping.

## CReverbBuffer

CReverbBuffer defines an object that is an EAX interface for a secondary buffer—a sound source. It works through the 3D interface of the secondary sound source.

### Constructor

```
CReverbBuffer(LPDIRECTSOUND3DBUFFER p3DBuf)
```

The constructor takes a pointer to the 3D interface of the 3D secondary buffer you want to set.

### Public Methods

```
int PropertySetOk()
```

This method checks to see if the audio driver supports the EAX sound-source property set. When called, it returns a TRUE if EAX is supported, a false if not.

```
void SetReverbMix(float mix)
```

This method sets the Reverb Mix listener property. It accepts a floating point value from 0.0 to 1.0. (See “Reverb Mix Property” on page 24 for details on what this value sets.)

```
void SetAll(EAXBUFFER_REVERBPROPERTIES *pProperties)
```

This method sets all the sound-source properties at one time. It accepts a pointer to an array of the property values you want to set. Because there’s only a single property in the sound-source property set at present, there’s no need to use this method.

```
float GetReverbMix()
```

This method returns the current Reverb Mix property value.

```
void GetAll(EAXBUFFER_REVERBPROPERTIES *pProperties)
```

This method returns a pointer to an array of all the current sound-source property values. Because there’s only a single property in the sound-source property set at present, there’s no need to use this method.

## Instantiating an EAX Class

You can create an EAX class using the class's constructor and a pointer to the 3D interface of a secondary buffer. This example creates a CReverb object that works through the 3D interface of secondary buffer 3Dbuf[0]:

```
#include "creverb.h"
CReverb *pReverb;
...
pReverb=new CReverb(p3Dbuf[0]);
if (!pReverb->PropertySetOk())
    AfxMessageBox("EAX not supported");
```

Notice that this code sample calls PropertySetOK on the newly created CReverb object to make sure that the audio driver supports EAX. Here's a code sample that creates a CReverbBuffer object for each secondary buffer created by a program:

```
#include "crevbuf.h"
CReverbBuffer *pReverbBuffer[n];
...
pReverbBuffer[i]=new CReverbBuffer(p3Dbuf[i]);
if (!pReverbBuffer[i]->PropertySetOk())
    AfxMessageBox("EAX not supported");
```

## Setting Individual Properties

Once you've created an EAX interface using CReverb or CReverbBuffer, you can use it to set individual properties for either the listener (through CReverb) or a sound source (through CReverbBuffer). The following four calls set the four listener properties through the CReverb object created in the previous section:

```
pReverb->SetEnviroment(EAX_ENVIROMENT_AUDITORIUM);
pReverb->SetVolume(0.5F);
pReverb->SetDecayTime(4.0F);
pReverb->SetDamping(0.5F);
```

The following statement sets the Reverb Mix property of a single secondary buffer through the CReverbBuffer object created in the previous section:

```
pReverbBuffer[i]->SetReverbMix(1.0F);
```

## Setting All Properties at One Time

If you want to set values for all four listener properties at once, you can do so by calling SetAll and passing it a pointer to an array of four values:

```
EAX_REVERBPROPERTIES params = { EAX_ENVIROMENT_AUDITORIUM,
    0.5F,4.0F,0.5F};
pReverb->SetAll(&params);
```

## Setting a Preset

To set all the listener properties at once by using a preset, call `SetPreset`:

```
pReverb->SetPreset(EAX_PRESET_SMALL_AUDITORIUM);
```

`EAX_PRESET_SMALL_AUDITORIUM` is a `#define` in a header file that expands into a set of environment properties, so the preceding call is equivalent to:

```
pReverb->SetPreset(EAX_ENVIROMENT_AUDITORIUM,0.5F,4.0F,0.5F);
```

if, of course, these are the four values defined by `EAX_PRESET_SMALL_AUDITORIUM`.

# EAX Properties

EAX contains two property sets: the sound-source property set and the listener property set. This section describes the properties of each set.

## Sound-Source Property Set

The sound-source property set contains a single property that applies through a property-set interface to a secondary buffer. To use this property, you must specify the property-set GUID

**DSROPERTYID\_EAXBUFFER\_ReverbProperties.**

### Reverb Mix Property

<b>Specify using this ID</b>	DSROPERTY_EAXBUFFER_REVERBMIX
<b>Value type</b>	Float
<b>Value range</b>	0.0 to 1.0
<b>Default value</b>	EAX_REVERBMIX_USEDISTANCE
<b>Value units</b>	A linear multiplication value

In a 3D aural world, the ratio of reverb to the sound source (called the *wet/dry* ratio) is usually set by a sound source's distance from the listener. If the source is close, the wet/dry ratio is low—the sound source (*dry*) is close to the ear, so you hear a lot of it. The reverb (*wet*) is reflected back from surrounding walls, so it's significantly less than the source sound. As the source moves away from the listener, the dry component diminishes with distance while the wet component remains approximately the same—so the wet/dry ratio goes up. The rule of thumb is: the greater the distance between source and listener, the higher the wet/ dry ratio.

If an application creates a 3D secondary buffer and doesn't set Reverb Mix, EAX automatically calculates the wet/dry ratio for this sound source, increasing or decreasing the wet value to create a very realistic 3D effect. (DirectSound itself increases or decreases the source's volume (the dry value) automatically as the source moves toward or away from the listener.) Most applications never use Reverb Mix to override EAX's reverb mix management.

If an application does want to override EAX, it sets Reverb Mix, which directly controls the amount of reverb (wet) in the wet/dry ratio. The reverb value range goes from 0.0 (no reverb at all added to the source) to 1.0 (the maximum amount of reverb added to the source). Once the property is set, the wet value remains the same until the property is reset. If the sound source moves toward or away from the listener, the wet/dry ratio changes as DirectSound increases or decreases the source volume while the reverb volume remains the same. If you want to put Reverb Mix back under the control of EAX, you can set the value using the constant **EAX\_REVERBMIX\_USEDISTANCE**.



## Listener Property Set

The listener property set contains four properties that apply through a secondary-buffer property-set interface to the primary buffer. To use these properties, you must specify the property-set GUID **DSPROPSETID\_EAX\_ReverbProperties**.

### Environment Property

<b>Specify using this ID</b>	DSPROPERTY_EAX_ENVIRONMENT
<b>Value type</b>	Unsigned long
<b>Value range</b>	0 to EAX_MAX_ENVIRONMENT
<b>Default value</b>	EAX_ENVIRONMENT_GENERIC
<b>Value units</b>	Integers that each specify a specific environment

Environment is the fundamental listener property. You typically set it first and then—if you want—modify it using the other three listener properties: Volume, Decay Time, and Damping.

When you set an environment, you choose the acoustic surroundings of the listener—the size of the virtual room around the listener and the reflective qualities of its walls. When you modify the environment using the other listener properties, you change overall reverb volume and the reflective quality of the walls. The room size remains the same.

The size of the virtual room controls the length of time it takes the first echo to come back from the walls—and hence the echo delay quality of the reverb. Consider, for example, a vast canyon. When you shout into it, it may take a second or two for the first echo to return; subsequent echoes follow it at similar intervals. If you shout in an acoustically live concert hall, the first echo comes back very quickly and is almost immediately awash in subsequent echoes, creating a warm reverb sound instead of a series of distinct echoes. Setting an environment with a large room size increases the time of the reverb's echo delay. The larger the room size, the more spacious the acoustic environment feels.

EAX defines each environment internally. It not only defines the room size and reflective wall qualities, it also defines the reverb model used for each environment. The driver that supports EAX may, in fact, use entirely different reverb algorithms for different environments.

To specify an environment, use an integer constant defined in the EAX.H file. Those environment constants are:

- EAX\_ENVIRONMENT\_GENERIC
- EAX\_ENVIRONMENT\_PADDEDCELL
- EAX\_ENVIRONMENT\_ROOM

- EAX\_ENVIRONMENT\_BATHROOM
- EAX\_ENVIRONMENT\_LIVINGROOM
- EAX\_ENVIRONMENT\_STONEROOM
- EAX\_ENVIRONMENT\_AUDITORIUM
- EAX\_ENVIRONMENT\_CONCERTHALL
- EAX\_ENVIRONMENT\_CAVE
- EAX\_ENVIRONMENT\_arena
- EAX\_ENVIRONMENT\_HANGAR
- EAX\_ENVIRONMENT\_CARPETEDHALLWAY
- EAX\_ENVIRONMENT\_HALLWAY
- EAX\_ENVIRONMENT\_STONECORRIDOR
- EAX\_ENVIRONMENT\_ALLEY
- EAX\_ENVIRONMENT\_FOREST
- EAX\_ENVIRONMENT\_CITY
- EAX\_ENVIRONMENT\_MOUNTAINS
- EAX\_ENVIRONMENT\_QUARRY
- EAX\_ENVIRONMENT\_PLAIN
- EAX\_ENVIRONMENT\_PARKINGLOT
- EAX\_ENVIRONMENT\_SEWERPIPE
- EAX\_ENVIRONMENT\_UNDERWATER
- EAX\_ENVIRONMENT\_DRUGGED
- EAX\_ENVIRONMENT\_DIZZY
- EAX\_ENVIRONMENT\_PSYCHOTIC

The name of each constant gives you a good idea of the environment's acoustic qualities: small room, large room, live surfaces, dead surfaces, and so forth.

**EAX\_ENVIRONMENT\_GENERIC**, environment 0, is the default Environment setting. It specifies an average room size of the best possible reverb quality.

Whenever you set the Environment property alone (that is, not as part of a set of four property values at once), EAX automatically sets the values of the other three listener properties to defaults for the specified environment.

The presets in the EAX.H file correspond one-to-one with the environment constants. Each preset shows the default property values defined by each environment; by reading the header file, you can learn those default values for each environment.

## Volume Property

<b>Specify using this ID</b>	DSPROPERTY_EAX_VOLUME
<b>Value type</b>	Float
<b>Value range</b>	0.0 to 1.0
<b>Default value</b>	Varies depending on the environment
<b>Value units</b>	A linear amplitude value

The Volume property is the master volume control for the reverb added to all sound sources; it sets the maximum volume of all reverb added to the sound mix in the primary buffer (the listener). Sound sources mixed into the primary buffer may have reverb volumes that vary relative to each other. (The reverb for each source is set by its wet/dry ratio as described in the Reverb Mix sound-source property.) As the master reverb volume goes up and down, controlled by Volume, the reverb for each sound source goes up or down in proportion.

The Reverb Volume property value is a linear amplitude value, not a linear decibel value. The maximum value of 1.0 turns overall reverb up to its fullest possible volume. Each time you halve the value, you halve the volume and therefore drop it by 6 dB. For example, turning the volume down from 1.0 to 0.5 drops the volume by 6 dB. Turning it down from 1.0 to 0.25 drops 12 dB; from 1.0 to 0.125 18 dB; and so on. Setting the volume to 0.0 turns off all reverb completely.

## Decay Time Property

<b>Specify using this ID</b>	DSPROPERTY_EAX_DECAYTIME
<b>Value type</b>	Float
<b>Value range</b>	0.1 to 20.0
<b>Default value</b>	Varies depending on the environment
<b>Value units</b>	Seconds

Reverb decay is caused when a room's surfaces absorb acoustic energy. Each time an echo bounces off a surface, it decreases in volume until there is no echo. If room surfaces are acoustically "live," they absorb very little acoustic energy, echoes diminish only gradually each time they bounce, and the reverb (a mixture of all the echoes) takes a long time to decay. If room surfaces are acoustically "dead," reverb decays very quickly as acoustic energy is absorbed.

The Decay Time property effectively sets the acoustic properties of the virtual room's surfaces by setting the time in seconds it takes the reverb to diminish by 60 dB. When set to the maximum value (20 seconds), it simulates very live surfaces; when set to the minimum value (0.1 seconds), it simulates very dead surfaces with almost no reverb at all.

The quality of the reverb decay set by this property is affected by the next property: Damping.

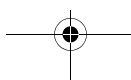
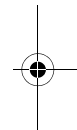
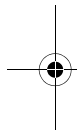


## Damping Property

<b>Specify using this ID</b>	DSPROPERTY_EAX_DAMPING
<b>Value type</b>	Float
<b>Value range</b>	0.0 to 2.0
<b>Default value</b>	Varies depending on the environment
<b>Value units</b>	A multiplier value

The acoustic reflectivity of a room surface is not always even across all frequencies. Some surfaces reflect more low and middle frequencies while absorbing high frequencies. This high-frequency roll-off is called *damping*.

The Damping property sets the amount of damping applied to an environment. Damping affects the decay time of the reverb as set by the property Decay Time. The damping value 1.0 is neutral: the decay time is equal for high frequencies and for low and middle frequencies. As the damping value increases above 1.0, the high-frequency decay time increases so it's longer than the decay time of the middle and low frequencies. You hear a more brilliant reverb. As the damping value decreases below 1.0, high-frequency decay time decreases so it's shorter than the decay time of the middle and low frequencies. You hear a more muffled reverb.



## Creating an EAX Object

The global reverb object (listener object) is created by calling the QueryInterface function on a secondary 3D buffer even though using that object will affect all 3D buffers. This may cause a potential problem if EAX settings are asserted after the 3D buffer has been released. To prevent this, reestablish the EAX listener object through another 3D buffer. It may be prudent to create a small dummy 3D buffer and keep that buffer open for the life of the game. The following sample code shows you how to create the listener object with this dummy buffer in mind:

```
#define PSET_SETGET (KSPORPERTY_SUPPORT_GET|KSPROPERTY_SUPPORT_SET)
// local small dummy buffer
static LPDIRECTSOUNDBUFFER pBuffer=NULL;    // dummy DS buffer
//-----//
//
//  EAXCreate
//
//  DESCRIPTION:
//    Opens an EAX environment if one is available
//
//  PARAMETERS:
//    pDS: direct sound object
//
//  RETURNS:
//    EAX object pointer, NULL if none can be found
//-----//
LPKSPROPERTYSET EAXCreate(LPDIRECTSOUND pDS)
{
    WAVEFORMATEX fmt={WAVE_FORMAT_PCM,2,44100,176400,4,16,0};
                                // for dummy 3D buffer
    DSBUFFERDESC desc;
    LPDIRECTSOUND3DBUFFER p3DBuf;    // 3D buffer interface
    LPKSPROPERTYSET pEAX;    // property set interface
    unsigned long support=0;    // variable to hold support status
    // Make sure previous dummy buffer has been released
    if (pBuffer)
    {
        IDirectSoundBuffer_Release(pBuffer);
        pBuffer=NULL;
    }
    // Create the dummy buffer
    memset(&desc, 0, sizeof(DSBUFFERDESC));
    desc.dwSize = sizeof(DSBUFFERDESC);
    desc.dwFlags = DSBCAPS_STATIC | DSBCAPS_CTRL3D;
    desc.dwBufferBytes = 128;
    desc.lpwfxFormat = &fmt;
    if (IDirectSound_CreateSoundBuffer(pDS,&desc,&pBuffer,NULL) != DS_OK)
        return NULL;
    // Create a 3D buffer interface
    if (IDirectSoundBuffer_QueryInterface(pBuffer,
        &IID_IDirectSound3DBuffer,&p3DBuf) != DS_OK
        || IDirectSound3DBuffer_QueryInterface(p3DBuf,&IID_IKsPropertySet,
        &pEAX) != DS_OK)
```

```

    {
        IDirectSoundBuffer_Release(pBuffer);
        pBuffer=NULL;
        return NULL;
    }

    // Create the EAX reverb interface
    if (IKsPropertySet_QuerySupport(pEAX,&DSPROPSETID_EAX_ReverbProperties,
        DSPROPERTY_EAX_ALL,&support) != DS_OK
        || (support & PSET_SETGET) != PSET_SETGET)
    {
        IDirectSoundBuffer_Release(pBuffer);
        pBuffer=NULL;
        pEAX=NULL;
    }
    return pEAX;
}

//-----//
//
//  EAXrelease
//
//  DESCRIPTION:
//    Closes an EAX environment
//
//  PARAMETERS:
//    pEAX: EAX object
//
//  RETURNS:
//    EAX object pointer, NULL if none can be found
//-----//
void EAXRelease(LPKSPROPERTYSET pEAX)
{
    if (pEAX)
        IKsPropertySet_Release(pEAX);
    if (pBuffer)
    {
        IDirectSoundBuffer_Release(pBuffer);
        pBuffer=NULL;
    }
}

```



## Future Plans for EAX

EAX is not a static development environment. We, at Creative, intend to constantly increase its capabilities to take advantage of advanced reverb properties offered by audio cards. Most of these advances will be transparent to programmers because they'll be incorporated into existing EAX environments. You use the environments as you always have; the EAX advancements simply make the environments sound even better.

We're also at work to turn EAX into an integrated component of DirectSound, which will allow developers to use EAX's reverb effects without going through a property-set interface. This also means that the same reverb effects will be simulated in software if an audio card doesn't exist to add them.

If you have ideas on future directions for EAX, please send them to **[mauchly@ensoniq.com](mailto:mauchly@ensoniq.com)**.

