Dynamic Audio

XNA Game Studio 4.0

# Content

[Content 2](#_Toc273493283)

[SoundEffect API in XNA 3.1 3](#_Toc273493284)

[Dynamic is Anti-Static 4](#_Toc273493285)

[New and Improved SoundEffect 6](#_Toc273493286)

[SoundEffect Constructors 6](#_Toc273493287)

[DynamicSoundEffectInstance 7](#_Toc273493288)

[Creating DynamicSoundEffectInstance 8](#_Toc273493289)

[Managing Playback 8](#_Toc273493290)

[Pull-Model Buffer Submission 8](#_Toc273493291)

[Push-Model Buffer Submission 9](#_Toc273493292)

[Managing Playback State 9](#_Toc273493293)

[Play 9](#_Toc273493294)

[Stop 10](#_Toc273493295)

[Pause and Resume 10](#_Toc273493296)

[Parametric Playback Control 10](#_Toc273493297)

[Microphone 10](#_Toc273493298)

[Enumeration of Microphones 10](#_Toc273493299)

[Capture Format 11](#_Toc273493300)

[Capture Buffer 11](#_Toc273493301)

[Getting Captured Data 11](#_Toc273493302)

[Event-Driven Capture 12](#_Toc273493303)

[Pull-Model Capture 12](#_Toc273493304)

[Handling Microphone Disconnection 12](#_Toc273493305)

[Tips and Tricks 14](#_Toc273493306)

[Converting from Bytes to Samples 14](#_Toc273493307)

[Handling Endian-ness 14](#_Toc273493308)

The XNA Sound Effect API was first introduced with XNA Game Studio 3.0 in October 2008. Since then there have been several releases and the API has undergone an incremental evolution. XNA Game Studio 4.0 introduces new Dynamic Audio features that add substantial power to an audio developer's arsenal. This article provides a comprehensive look at these new features and insight into the architecture and the intricacies of coding to the new API additions.

# SoundEffect API in XNA 3.1

Before diving too deeply into the new dynamic audio features, let's briefly look at the Sound Effect API architecture and behavior in XNA Game Studio 3.1. Until now, integrating sound effects into an XNA game involved two steps:

* **Compile audio assets** **with the game** – This involves adding sound files to the game's content project. The Content Pipeline then processes these files during game compilation, performing any necessary format conversions, compression, and sample rate conversions. The resultant XNB files contain all the information necessary to create a **SoundEffect** type at runtime.



* **Load and play sounds in the game** – The game loads the XNB files by using the **ContentManager** type that returns a **SoundEffect** object.



**SoundEffect** type is designed to optimize memory usage by allowing all multiple simultaneous playing instances to share the source audio data while still allowing each instance to have unique playback characteristics such as pitch, volume, pan, or 3D position, and looping control.

A game can use a *Fire and Forget* pattern to play back quick, maintenance-free instances.

SoundEffect gunfireSound;

protected override void LoadContent()

{

…

gunfireSound = Content.Load<SoundEffect>("PewPew");

…

}

private void FireGun()

{

// Simple "Fire and Forget" playback

gunfireSound.Play();

}

The *Create, Configure, and Play* pattern also can be used. This pattern provides better parametric control over the playback of an instance—for example, controlling looping and 3D position.

SoundEffect gunfireSound;

SoundEffectInstance gunfireInstance;

protected override void LoadContent()

{

…

gunfireSound = Content.Load<SoundEffect>("PewPew");

// Create gunfire instance

gunfireInstance = gunfireSound.CreateInstance();

gunfireInstance.IsLooped = true;

…

}

// Volume, pitch and pan is set based on game logic.

// For example depending on the enemy proximity, weapon used and spatial position.

private void FireGun(float volume, float pitch, float pan)

{

// Set the gunfire parameters based on enemy instance

gunfireInstance.Volume = volume;

gunfireInstance.Pitch = pitch;

gunfireInstance.Pan = pan;

gunfireInstance.Pitch = pitch;

gunfireInstance.Play();

}

This is a brief and high-level overview. The important point is that a game is restricted to using sounds defined at *design time*. Once loaded, audio data for a **SoundEffect** cannot be changed, effectively making these sounds *static*.

# Dynamic is Anti-Static

There are scenarios where constraining audio to static sounds can be limiting. For example, allowing a game to do some processing on the *dry* sounds before playback can reduce the download size and make the game audio more interesting. A game may also want to use procedurally synthesized sounds, or a custom file format with compression that performs some processing and decoding before playback. Under XNA, these scenarios are impossible without the new Dynamic Audio features. These new features allow games to create sound effects at *runtime* from raw buffers of audio samples.

XNA Game Studio 4.0 has several dynamic audio-related changes:

* New **SoundEffect** constructors that take format description and an audio buffer.
* New **DynamicSoundEffectInstance** type that allows playback of a stream of audio buffers.
* New **Microphone** type to capture audio from connected microphones.

Before examining these additions in greater detail, let's review two important concepts: audio format and block alignment.

* **Audio format** describes all the information necessary for the audio subsystem to take a bucket of bytes and generate a sound. XNA dynamic audio features expect the audio data to be 16-bit Integer-PCM, sample rate from 8000 Hz to 48,000 Hz, mono or stereo.
* An audio **block alignment** value describes the count of bytes necessary to produce the minimum number of whole audio samples for all encoded channels. Audio subsystems typically require that the data is aligned to this block alignment value. For PCM encoding, computing this value is straightforward:

Block Alignment = Bytes-Per-Sample \* Channels

Since the dynamic audio format is restricted to 16-bit samples, the value changes depending on the number of channels, mono or stereo.

Here is what an 8-byte buffer containing 16-bit PCM **mono** audio data looks like:



The block alignment value for mono PCM data is 2 bytes, the same as its bytes-per-sample value.

The same 8-byte buffer with 16-bit stereo data contains interleaved left and right samples.



The block alignment for stereo PCM data, therefore, works out to 4 bytes.

# New and Improved SoundEffect

Changes to the **SoundEffect** type include new constructors to create it from raw audio buffers and new methods that help translate between the duration of the sound and the size of the audio buffer.

## SoundEffect Constructors

public SoundEffect (byte[] buffer,int sampleRate,AudioChannels channels)

public SoundEffect (byte[] buffer,int offset,int count,int sampleRate,

AudioChannels channels,int loopStart,int loopLength)

By using these new constructors, the game can create sound effects from a raw audio buffer and continue to get all the benefits—like fire and forget playback and the ability to share audio memory between multiple instances—offered by the **SoundEffect** type. Although **SoundEffect** can now be created from dynamically generated audio data, the fact is that usage semantics of the **SoundEffect** type remain unchanged.

The specified buffer length, the offset, and the count must be block aligned. Remember, block alignment can change depending on the number of channels in the audio signal, 2 for mono and 4 for stereo. **SoundEffect** type provides two helper methods to convert between time units and buffer size. Using these methods guarantees that the values are always block aligned for the format.

public static int GetSampleSizeInBytes (TimeSpan duration,int sampleRate,

AudioChannels channels)

public static TimeSpan GetSampleDuration (int sizeInBytes,int sampleRate,

AudioChannels channels)

Here is a simple example that creates a **SoundEffect** from a raw buffer:

SoundEffect dynamicSoundEffect;

byte[] buffer;

protected override void CreateAndPlaySoundEffect()

{

// Allocate a buffer big enough to hold 500ms of stereo audio data at 48000Hz

buffer = new byte[SoundEffect.GetSampleSizeInBytes(TimeSpan.FromMilliseconds(500),

48000, AudioChannels.Stereo)];

// Fill the buffer with an audio signal

FillBufferWithSineWave(buffer);

// Create a SoundEffect from this buffer

dynamicSoundEffect = new SoundEffect(buffer, 48000, AudioChannels.Stereo);

// Quickly play the sound using fire and forget playback

dynamicSoundEffect.Play();

}

# DynamicSoundEffectInstance

Seen strictly in terms of the type hierarchy, a **DynamicSoundEffectInstance** *is* a specialized **SoundEffectInstance**. It inherits from a **SoundEffectInstance** and, in some fundamental ways, behaves exactly like a **SoundEffectInstance**. **DynamicSoundEffectInstance** provides all the same semantics to control the playback state (Play, Pause, Stop), as well as allowing parametric control (Pitch, Pan, Apply3D, Volume) of the sound. In other ways though, a **DynamicSoundEffectInstance** is a different beast than **SoundEffectInstance**.



Unlike a **SoundEffectInstance** that relies on a **SoundEffect** for its creation and *always* references the audio data owned by this parent **SoundEffect**, a **DynamicSoundEffectInstance** works independently and manages its own internal queue of audio buffers. When a game adds buffers to this queue, the **DynamicSoundEffectInstance** dutifully plays them back in the order they are received.

## Creating DynamicSoundEffectInstance

public DynamicSoundEffectInstance (int sampleRate, AudioChannels channels)

At creation time, a **DynamicSoundEffectInstance** only needs to know the format in which it needs to render audio. The format for dynamic audio features in XNA currently is limited to 16-bit PCM; the only configurable parts of the format are the sample rate (8000 Hz to 48,000 Hz), and the number of channels (mono or stereo). **DynamicSoundEffectInstance** uses this format information to reserve an audio voice—the low-level audio engine resource that actually renders the audio.

## Managing Playback

**DynamicSoundEffectInstance** provides a **SubmitBuffer** method that allows the game to queue a buffer for playback.

public void SubmitBuffer (byte[] buffer)

public void SubmitBuffer (byte[] buffer,int offset,int count)

The two overloads allow the game to playback the entire buffer or a subregion within a larger buffer. The buffer length, offset, and count values must be block aligned. **DynamicSoundEffectInstance** provides the same **GetSampleSizeInBytes** and **GetSampleDuration** methods that simplify this calculation. On buffer submission, **DynamicSoundEffectInstance** copies the audio data from the passed buffer into its internal queue. This means that the game is free to operate on this buffer as soon as the **SubmitBuffer** call returns. The **PendingBufferCount** property returns the count of buffers currently in the queue, including the buffer currently playing.

**DynamicSoundEffectInstance** plays back the queued buffers sequentially. If it runs out of buffers to play, it produces silence. In cases where a game needs to produce a constant audio stream, this means **DynamicSoundEffectInstance** must regularly submit the audio buffers to prevent the queue from running out. If this is handled incorrectly, audio playback can produce audible glitches, which leads to a poor experience.

There are several means available to ensure buffer submission occurs correctly and to avoid any glitches.

### Pull-Model Buffer Submission

**DynamicSoundEffectInstance** provides an *event-driven* mechanism that alerts the game when more audio data needs to be submitted to avoid glitches. To do this, a game subscribes to the BufferNeeded event raised when the pending buffer count falls below 3, and until it reaches 0. Once the buffer count reaches 0, **DynamicSoundEffectInstance** assumes that the client is not interested in immediately submitting more buffers and stops raising this event. Of course, if the game queues more buffers, the event is raised again as the queued buffers are consumed.

### Push-Model Buffer Submission

In this model, the game handles the timing of buffer submission on its own by monitoring the **PendingBufferCount** property. This mechanism is useful for low-latency scenarios where the game needs to queue small number of short buffers that can change rapidly.

The following is an example of how the event-driven buffer submission works:

DynamicSoundEffectInstance dynamicSoundInstance;

byte[] buffer;

protected override void CreateDynamicSoundEffectInstance()

{

// Create a DynamicSoundEffectInstance that can play 48KHz, stereo data.

dynamicSoundInstance = new DynamicSoundEffectInstance(48000, AudioChannels.Stereo);

// Allocate a buffer big enough to hold 100ms worth of audio data.

buffer = new byte[dynamicSoundInstance.GetSampleSizeInBytes(

TimeSpan.FromMilliseconds(100))];

// Fill the buffer with some audio data

FillBufferWithSineWave(buffer);

// Subscribe to the BufferNeeded event for event-drive buffer submission

dynamicSoundInstance.BufferNeeded +=

new EventHandler<EventArgs>(DynamicSoundInstance\_BufferNeeded);

// Start the sound, no currently pending buffers but sound will raise

// the BufferNeeded event giving us a chance to submit buffers for playback.

dynamicSoundInstance.Play();

}

void DynamicSoundInstance\_BufferNeeded(object sender, EventArgs e)

{

// Submitting the larger buffers as two smaller buffers ensures that

// the event is raised before the buffers run out, avoiding glitches

dynamicSoundInstance.SubmitBuffer(buffer, 0, buffer.Length/2);

dynamicSoundInstance.SubmitBuffer(buffer, buffer.Length/2, buffer.Length/2);

}

## Managing Playback State

**DynamicSoundEffectInstance** inherits from **SoundEffectInstance**, and provides the same state control semantics. There are, however, behavioral differences between the two that are important to understand.

### Play

When **Play** is called, **DynamicSoundEffectInstance** immediately starts playing back the pending buffers in the queue. If the queue is empty, it immediately raises the BufferNeeded event, giving the game a chance to start submitting buffers. Of course, if the game does not submit any buffers, **DynamicSoundEffectInstance** does nothing—effectively producing silence—until buffers are added to the queue.

### Stop

The behavior of the **Stop** call depends on which overload of **Stop** is used. The default **Stop** behavior results in an immediate stop; all pending buffers in the queue are flushed immediately. Conversely, with a non-immediate **Stop** (using Stop(false) ), playback continues until all buffers in the queue are consumed and the end of stream is reached.

### Pause and Resume

**Pause** and **Resume** calls behave exactly as they do for **SoundEffectInstance** type.

## Parametric Playback Control

**DynamicSoundEffectInstance** inherits all the parametric controls from the **SoundEffectInstance** type. The only difference is that **DynamicSoundEffectInstance** does not provide built-in looping support. A game can easily handle looping on its own by queuing buffers as needed. **DynamicSoundEffectInstance** playback is sample accurate, and will not produce any undesirable artifacts as long as the queue does not run out of buffers.

# Microphone

Along with the ability to render raw audio samples, XNA Game Studio 4.0 introduces a way to *capture* audio samples from a connected microphone.



The design of the **Microphone** type is complimentary to that of **DynamicSoundEffectInstance**. It provides semantics that are similar to **DynamicSoundEffectInstance** and allows an easy intuitive interaction between the two. **Microphone** makes it easy to enumerate the connected microphones, to do some basic configuration, to control the state, and to acquire the captured audio data.

## Enumeration of Microphones

XNA Framework's capture stack actively tracks the state for all connected microphone devices. **Microphone** type provides two properties that make enumerating connected microphones easy. A static **All** property returns read-only collection of all available microphones. A game can simply remove a reference to a microphone from this collection to use it. An even simpler way is to use the **Default** property, which returns the first good microphone without requiring the game to go through the collection.

Here is additional information about enumerating microphones:

* The order of microphones in the collection is constant and preserved across connection and disconnection of actual devices. On microphone device disconnection, the corresponding managed **Microphone** object continues to be valid, but any attempt to use this microphone will throw a **NoMicrophoneConnectedException** (more about this later).
* Default microphone will not change during the lifetime of the game.
* New microphones always get appended to the end of the list. This means that it is safe to not reference the collection using an index. The **Microphone** instance at that index will not change underneath the game.

## Capture Format

The capture stack ensures that all microphones conform to the same basic audio format and return 16-bit PCM mono audio data. The only variable in the format is the sample rate, which can change between different types of microphones. Games should always use the **SampleRate** property to discover this value.

## Capture Buffer

The low-level audio capture stack uses an internal circular buffer to acquire the audio data from the actual microphone device. The size of this buffer is configurable via the **BufferDuration** property. This buffer is double-buffered, and the actual size of the buffer is twice what is specified by the **BufferDuration** property. This is a circular buffer and a game must pick up any available audio data within (2 \* BufferDuration) time. The size constraint on the **BufferDuration** requires that it must be between 100 milliseconds (ms) and 1 second, and 10 ms aligned. Correspondingly, the low-level capture buffer size can vary between 200 ms minimum and 2 seconds maximum.

## Getting Captured Data

Microphone provides a **GetDat**a method that returns the captured audio data.

public int GetData (byte[] buffer)

public int GetData (byte[] buffer,int offset,int count)

In both **GetData** variants, the buffer length, offset, and count values must be block aligned. The block alignment value for **Microphone** is always 2 since it always returns 16-bit PCM mono data. Again, **Microphone** provides the **GetSampleSizeInBytes** and **GetSampleDuration** helpers to make this easy. The method returns the size in bytes of the actual captured audio data copied to the buffer. In cases where there is no new capture data, this method returns 0; for example, when **GetData** is called faster than the granularity of the low-level capture engine.

In a pattern similar to **DynamicSoundEffectInstance**, **Microphone** provides couple different ways to acquire the captured data.

### Event-Driven Capture

In a pattern analogous to the **DynamicSoundEffectInstance** pull-model submission, the **Microphone** type provides a BufferReady event that is raised when a **BufferDuration** worth of capture data is available since the last **GetData** call. By configuring the **BufferDuration** property, the game can control how often this event is raised. **Microphone** type guarantees that there is at least a **BufferDuration** worth of capture data ready to be consumed when this event is raised. The game has another **BufferDuration** to pick up the data without causing a glitch.

### Pull-Model Capture

In this model, the game calls **GetData** at its own convenience. While event-driven capture has a minimum latency of 100 ms, the pull-model capture can be a much lower latency controlled by the game, which makes this pattern desirable for certain scenarios like Karaoke games.

The following is an example of an event-driven capture:

byte[] buffer;

private void SetupAndStartMicrophone()

{

// No microphone connected?

if (Microphone.Default == null)

{

return;

}

// Microphone will call us back everytime it has 100ms data ready.

// Low-level capture buffer will be 200ms.

Microphone.Default.BufferDuration = TimeSpan.FromMilliseconds(100);

// Subscribe to the BufferReady event.

Microphone.Default.BufferReady += new EventHandler<EventArgs>(Default\_BufferReady);

// Allocate a buffer big enough to hold 200ms worth of data.

buffer = new byte[2 \* Microphone.Default.GetSampleSizeInBytes(

Microphone.Default.BufferDuration)];

}

void Default\_BufferReady(object sender, EventArgs e)

{

int recorded = Microphone.Default.GetData(buffer);

ProcessCapturedData(buffer, recorded);

}

## Handling Microphone Disconnection

Any microphone device, for example a USB microphone, can be disconnected at any time. **Microphone** type does not support an **IsConnected** property because it cannot be guaranteed to be accurate. Instead, **Microphone** throws a **NoMicrophoneConnectedException** when an attempt is made to use a disconnected microphone. A game should handle this exception and prompt the user to reconnect the microphone. For simple scenarios, the game can implement an **IsConnected** extension method that converts the exception to a Boolean value.

public static class MicrophoneExtensions

{

// Provides a simple method to check if a microphone is connected.

// There is no guarantee that the microphone will not get disconnected at right

// after this method returns but it can help in simplifying the microphone

// enumeration code.

public static bool IsConnected(this Microphone microphone)

{

try

{

MicrophoneState state = microphone.State;

return true;

}

catch (NoMicrophoneConnectedException)

{

return false;

}

}

}

private Microphone PickFirstConnectedMicrophone()

{

// Let's pick the default microphone if it's ready.

if (Microphone.Default != null && Microphone.Default.IsConnected())

{

return Microphone.Default;

}

// Default microphone seems to be disconnected so look for another microphone that we

// can use. If the default was null then the list will be empty and we'll skip the

// search.

foreach (Microphone microphone in Microphone.All)

{

if (microphone.IsConnected())

{

return microphone;

}

}

// There are no microphones hooked up to the system!

return null;

}

# Tips and Tricks

## Converting from Bytes to Samples

While the Dynamic Audio features restrict the format to 16-bit PCM, the API works with arrays of bytes. Why? Wouldn't it be simpler if the API just handled shorts? Not really. The API is designed with multiple principles. It needs to provide a familiar usage pattern, interface well with other .Net APIs that take buffers like streams, and it must be ready for extensibility for future format support. This means that there is some extra work necessary especially when doing any audio processing to convert from bytes to 16-bit audio samples, shorts. This conversion also needs to account for the endian-ness of the platform the game is running on.

## Handling Endian-ness

The SoundEffect API is completely cross-platform and works without requiring any changes on Windows, Xbox 360, and Windows Phone 7. Managing audio data on the other hand requires a little more work depending on the endian-ness of the platform. Xbox 360 expects audio data to be big-endian; a game should take that into account. By default, **Microphone** on Xbox 360 returns the data in big-endian format. If the game is passing the data straight through for playback, it does not need to do anything. For processing, the game may need to byte-swap the data when converting between bytes and shorts. The following shows how this can be achieved:

// Returns a sample value from the passed buffer,

// taking into account the endian-ness of the system.

private short ReadSample(byte[] buffer, int index)

{

// Ensure we're doing aligned reads.

if (index % sizeof(short) != 0)

{

throw new ArgumentException("index");

}

if (index >= buffer.Length)

{

throw new ArgumentOutOfRangeException("index");

}

short sample = 0;

if (!BitConverter.IsLittleEndian)

{

sample = (short)(buffer[index] << 8 | buffer[index + 1] & 0xff);

}

else

{

sample = (short)(buffer[index] & 0xff | buffer[index + 1] << 8);

}

return sample;

}

// Writes the sample value to the buffer,

// taking into account the endian-ness of the system.

private void WriteSample(byte[] buffer, int index, short sample)

{

// Ensure we're doing aligned writes.

if (index % sizeof(short) != 0)

{

throw new ArgumentException("index");

}

if (index >= buffer.Length)

{

throw new ArgumentOutOfRangeException("index");

}

if (!BitConverter.IsLittleEndian)

{

buffer[index] = (byte)(sample >> 8);

buffer[index + 1] = (byte)sample;

}

else

{

buffer[index] = (byte)sample;

buffer[index + 1] = (byte)(sample >> 8);

}

}