DFVoice Developer Docs

Thank you for purchasing DFVoice, a plugin by DaikonForge. By using DFVoice, you're empowering your players with powerful real time communication built on industry standard and battle-tested audio compression techniques.
DFVoice is designed to be network agnostic, allowing you to integrate with nearly any multiplayer solution to suit your needs. This requires a bit of "glue" code from you to get communications up and running. Luckily, this is quite a simple matter.

Getting Started

Let's start by replicating the included Local Demo.
Create a new script MyLocalVoiceController and make the new class inherit from VoiceControllerBase.

```
using UnityEngine;
using System.Collections;

using DaikonForge.VoIP;

public class MyLocalVoiceController : VoiceControllerBase
{
}
```

This class is responsible for integrating DFVoice with your network solution. First, we need to implement the IsLocal property.
DFVoice is designed such that there is one voice controller object per client. The IsLocal property is responsible for telling DFVoice which of these actually belongs to the local client (and therefore should record and send voice data). Since this is a local-only demo, we'll just return true.

```
public class MyLocalVoiceController : VoiceControllerBase
{
    public override bool IsLocal
```

```
    {
        get { return true; }
    }
}
```

Next, we'll need to actually "send" the voice data. We'll start by overriding the OnAudioDataEncoded function. This function is called every time a single "frame" of audio is encoded and ready for transmission (a "frame" is a single chunk of audio. the size of a frame is determined by a codec - in Speex, it's approximately 20ms of audio)

```
protected override void OnAudioDataEncoded( VoicePacketWrapper
encodedFrame )
{
}
```

Your code is responsible for calling the "ReceiveAudioData" function when it receives audio data over the network (this function tells DFVoice to decode and buffer the received audio to be played)
Since this is a local demo, we'll just directly call receiveAudio

```
protected override void OnAudioDataEncoded( VoicePacketWrapper
encodedFrame )
{
    ReceiveAudioData( encodedFrame );
}
```

Now our component is ready to be added to an object.
Create an empty game object in your scene and attach your new component. Additionally, add the Unity Audio Player (Component → DFVoice → Unity Audio Player) and the Microphone Input Device (Component → DFVoice → Microphone Input Device) components (more on these later).
One last thing before you listen to your own voice: make sure Debug Audio is enabled on your voice controller component. If Debug Audio is enabled, your own voice will be played back to you and is meant to allow you to test your setup and ensure that it works without needing a second machine. Otherwise, the local client's voice will not be played back to them (since that would be very distracting for reasons having something to do with science).
Now play the scene and speak into the microphone, you should hear your voice played back. Make sure your main microphone is set as the default device in your OS settings. If not, make it default and restart Unity (it is possible to change the microphone at runtime, but we'll cover that later).

Networking

Playing back voice locally is one thing, but sending it over the network is another thing. The idea is that you send three main pieces of data which define a frame: the frame index, the frequency ID, and the raw data.
The frame index (*encodedFrame.Index*) is an incrementing unsigned long value, used to determine if frames of data were lost - the codec is responsible for turning lost frames into some kind of “filler” audio (simpler codecs might simply return empty filler data or “comfort noise”, while Speex has some interpolation and decay techniques to help mask the missing data)
The frequency ID (*encodedFrame.Frequenc*y) is a one byte value used to determine the frequency of the encoded audio.
Finally, the raw data (*encodedFrame.RawData*) is a byte array which represents the encoded data ready to be sent over the network.

In some cases your networking solution can send these values as-is. However, some do not have built-in serialization solutions for ulongs and single bytes (Unity Networking for example). In these cases, we can generate another byte array for these header values (frame index and frequency values). DFVoice includes memory-friendly methods for performing this operation:

```
byte[] headers = encodedFrame.ObtainHeaders();

// send voice data here...

encodedFrame.ReleaseHeaders();
```

Now, on the receiving end, we need to reconstruct the frame and pass it to the ReceiveAudioData function. If you have the ulong and byte values, you can construct a new VoicePacketWrapper like this:

```
VoicePacketWrapper packet = new VoicePacketWrapper( index, frequency,
rawData );
```

Or, if you use the header byte array, you can do this:

```
VoicePacketWrapper packet = new VoicePacketWrapper( headers, rawData
);
```

And then pass this to the ReceiveAudioData function.

To demonstrate these concepts, here’s example code for a Unity Network voice controller implementation:

```
public class UnityNetworkVoiceController : VoiceControllerBase
```

```
{
    public override bool IsLocal
    {
        get{ return networkView.isMine; }
    }

    protected override void OnAudioDataEncoded( VoicePacketWrapper
encodedFrame )
    {
        byte[] headers = encodedFrame.ObtainHeaders();
        networkView.RPC( "vc", RPCMode.All, headers,
encodedFrame.RawData );
        encodedFrame.ReleaseHeaders();
    }

    [RPC]
    void vc( byte[] headers, byte[] rawData )
    {
        VoicePacketWrapper packet = new VoicePacketWrapper( headers,
rawData );
        ReceiveAudioData( packet );
    }
}
```

Switching Microphones

It is possible to switch microphones while the game is running. There are two methods of this - one, by setting the default microphone which will be used when the voice controller is started as a static variable, and another by calling the ChangeMicrophoneDevice method of the microphone component.
Both of these take a device name string which corresponds to the string value in the Microphone.devices array.
In the first case, this would be called prior to instantiating a voice controller object to set the default microphone which will be used.

```
MicrophoneInputDevice.DefaultMicrophone = "device name";
```

In the second case, this would be called while the game is running to switch recording to a new audio device.

```
MicrophoneInputDevice.ChangeMicrophoneDevice( "device name" );
```

Note that 'null' is equivalent to using the default microphone, which is Microphone.devices[0].

Push To Talk

The Microphone Input Device component supports push-to-talk. By default the push to talk key trigger is None, meaning voice is always sent. However, you can change it to any key you wish to enable Push To Talk mode (meaning the key must be held down in order for voice data to be sent)

Equalizing Audio

The built-in Unity Audio Player component supports audio equalization.
If enabled, it attempts to modify input audio to match a target maximum volume by applying a gain factor.
If the input sound is quiet, the gain factor increases over time (slowly raising the volume of the audio). If the input sound is loud, the gain factor is immediately decreased (in order to avoid sudden loud noises).
The inspector properties for this are as follows:

- The *Equalize* checkbox determines whether equalization will be applied.
- *Equalize Speed* is the speed at which gain factor is increased (at a rate of Time.deltaTime * EqualizeSpeed)
- *Target Equalize Volume* is the volume which audio equalization will attempt to match.
- *Max Equalization* is the maximum gain which will be applied. Without a maximum, the background noise would be amplified significantly to achieve the target volume - so a maximum is applied to limit this effect.

3D Sound

The built-in Unity Audio Player component also supports 3D audio playback. This makes the sound play just as any other 3D sound in Unity, from the position of the voice controller object. It supports reverb zones, distance attenuation, panning, doppler, and more. It can be very useful for some games, such as open-world survival games.
To enable 3D sound, simply tick the Is Three Dimensional checkbox of the Unity Audio Player component. If you want the sound to play from the position of a specific character, you'll need to place the voice controller at that character's position.

Architecture

In order to better understand DFVoice, let's take a look at the general flow of information

- First the Microphone component raises an event when audio data is available to be encoded.
- The voice controller handles this event by pushing the audio data to the codec and retrieving encoded frames. It calls OnAudioDataEncoded for each frame returned.
- OnAudioDataEncoded sends audio over the network
- The audio data is received and passed to the voice controller's ReceiveAudioData function.
- The voice controller passes the encoded data through the codec and retrieves decoded audio data.

- This audio data is passed to the Unity Audio Player component to be buffered and played.

This involves at least three components - the microphone, the voice controller, and the audio player.
All three of these are modular in nature, meaning it is possible to write a custom audio input device, voice controller, and audio output device.

Web Player Authorization

On webplayer platforms, you must ask for permission from the user to use their microphone before DFVoice will work. It is best to do this when the game first starts (from the main menu, or the splash screen)
How to do this is covered in the Unity documentation here.
DFVoice does not automatically request microphone access, but it will check to see if microphone access has been granted before attempting to start a recording (if microphone access has not been granted, the included Microphone Input Device will refuse to start a recording and will log a warning message)

.NET Version Compatibility

When building for some platforms such as iOS, you may need to set your Unity project's .NET compatibility to ".NET 2.0" instead of ".NET 2.0 Subset", otherwise you may get an error saying that NSpeex.DLL failed to cross-compile.